DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

A NEW APPROACH TO IP LEVEL CONGESTION CONTROL

by
Gökhan ÇATALKAYA

December, 2010
İZMİR

A NEW APPROACH TO IP LEVEL CONGESTION CONTROL

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy in Electrical & Electronics Engineering,
Electrical & Electronics Engineering Program

by
Gökhan ÇATALKAYA

December, 2010
İZMİR

Ph.D. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"A NEW APPROACH TO IP LEVEL CONGESTION CONTROL"** completed by **GÖKHAN ÇATALKAYA** under supervision of **PROF.DR. MUSTAFA GÜNDÜZALP** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

____

Prof. Dr. Mustafa GÜNDÜZALP
Supervisor

____

Assist. Prof. Dr. Adil ALPKOÇAK
Thesis Committee Member

____

Assist. Prof. Dr. Zafer DİCLE
Thesis Committee Member

____

Prof. Dr. M. Ertuğrul ÇELEBİ
Examining Committee Member

____

Ins. Dr. M. Kemal ŞİŞ
Examining Committee Member

____

Prof. Dr. Mustafa SABUNCU
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to express my appreciation to my advisors Prof. Dr. Mustafa GÜNDÜZALP and Dr. M. Kemal ŞİŞ for their advices, guidance and encouragement during my thesis.

I would like to thank to Assist. Prof. Dr. Adil ALPKOÇAK and Assist. Prof. Dr. Zafer DİCLE as the member of my thesis and thesis examination committee, and Prof. Dr. M. Ertuğrul ÇELEBİ as the member of my thesis examination committee, for their contribution, guidance and support.

And the last but not the least, my deepest thanks and love go to my parents, my wife Ebru ÇATALKAYA and my son Sarp ÇATALKAYA for their faithful encouragement and invaluable support and encouraging me during my life.

I would like to dedicate this thesis to my family.

Gökhan ÇATALKAYA

A NEW APPROACH TO IP LEVEL CONGESTION CONTROL

ABSTRACT

Due to the fast growth of the demand for the use of internet during the last decade, congestion control mechanisms to keep the throughput high and the average queuing delays low get of vital importance. In this thesis, the usage of the congestion control strategies in the growing world of networking is investigated. The purpose of this thesis is to present a new approach, which is called as “Orange” in IP level congestion control as an active queue management mechanism and to compare its performance of our proposed algorithm with that of the other mechanisms. Within the framework of this thesis, the best operating point of Orange algorithm is evaluated by using the empirical formulas we derive. It is investigated that when the best operating point parameters are applied, Orange gives the best performance among other active queue management algorithms.

In the context of this thesis, a general procedure for constructing threshold control policies that are implementable is described; and computer simulation is used to show that these policies perform well, especially in congestion conditions. The results obtained from the computer simulation are also used to justify the congestion reducing routing strategy approach.

The key observation shows that a good routing strategy that prevents servers from idling and wasting resource capacity is required for the networks when there is substantial work in the system.

Keywords: Queuing theory, congestion avoidance, congestion control, routing, threshold, computer simulation, active queue management.

IP SEVİYESİNDE TIKANIKLIK DENETİMİNE YENİ BİR YAKLAŞIM

ÖZ

Son yıllarda internet kullanımına olan talebin hızlı artışının etkisiyle, ağ verimliliğini yüksek ve ortalama kuyruk gecikmelerini düşük tutan tıkanıklık denetim yöntemleri önem kazanmıştır. Bu tez çalışmasında sürekli büyüyen bilgisayar ağları dünyasındaki tıkanıklık denetim yöntemlerinin kullanımını incelenmiştir. Bu tezin amacı ip seviyesinde tıkanıklık denetimine "Orange" adını verdiğimiz aktif kuyruk yönetimi olan yeni bir yaklaşım önermek ve önerdiğimiz bu yöntemin başarımını diğer yöntemler ile karşılaştırmaktır. Bu tezin kapsamında, geliştirdiğimiz ampirik formüller kullanılarak Orange algoritmasının en iyi çalışma parametreleri ölçümlendirilmiştir. Bu çalışma parametreleri uygulandığında, Orange algoritmasının diğer aktif kuyruk yönetimi algoritmaları arasında en iyi sonuçları verdiği gözlemlenmiştir.

Bu tezin içeriğinde, uygulanabilir eşik değerli denetim yöntemlerinin kurulması için genel bir yöntem tasarlanmış ve programladığımız bilgisayar benzetimi bu tür yöntemlerin özellikle ağır trafik şarlarında iyi çalıştığını göstermek için kullanılmıştır. Bilgisayar benzetiminden elde edilen sonuçlar çalışmamızda incelenen tıkanıklık azaltan iletim yöntemini doğrulamakta da kullanılmaktadır.

Sistemde büyük ölçüde iş olduğunda, sunucuların boş kalmasını ve kaynak kapasitesinin boşa harcanmasını engelleyen iyi bir iletim yönteminin, ağlar için gerekli olduğu ortaya çıkmıştır.

Anahtar Kelimeler: Kuyruk teoremi, tıkanıklık önleme, tıkanıklık denetimi, yönlendirme, eşik değer, bilgisayar benzetimi, aktif kuyruk yönetimi.

CONTENTS

Page

Ph.D. THESIS EXAMINATION RESULT FORM ii

ACKNOWLEDGEMENTS iii

ABSTRACT iv

ÖZ………… v

CHAPTER ONE - INTRODUCTION 1

1.1 General 1

1.2 The Evolving Internet 2

1.3 The Area of Research 4

1.4 Objectives and Scope 6

1.5 Overview of the Thesis 7

CHAPTER TWO - BASICS OF CONGESTION CONTROL 9

2.1 Overview 9

2.2 Congestion Collapse 9

2.3 Fairness 12

2.4 Flow Control 13

2.5 Additive Increase Multiplicative Decrease 14

2.6 Overview of TCP's Congestion Control 15

2.6.1 TCP Tahoe and Reno 17
2.6.2 Fast Retransmit 18
2.6.3 Fast Recovery 18
2.6.4 TCP Vegas 19
2.6.5 TCP New Reno 19
2.7 Classification of Congestion Control 20
2.8 RTT Estimation 22

CHAPTER THREE - BASICS OF QUEUEING SYSTEMS 24

3.1 General 24
3.2 The Arrival Process and the Queue 25
3.3 The Service Process and the Server 26
3.4 Queuing Discipline 26
3.5 Probability Distribution of Arrival or Service Times 27
3.5.1 Poisson Distribution 27
3.5.2 Exponential Distribution 28
3.5.3 Gamma (Erlang) Distribution 29
3.6 Notation for Queuing Systems 30
3.7 Queues and Probability Theory 31
3.8 Birth-death Markov Chains 35

CHAPTER FOUR - LITERATURE REVIEW 38

4.1 Congestion Avoidance Mechanisms 38

4.2 Scheduling Algorithms.. 39

4.3 Active Queue Management Algorithms.. 42

4.4 Explicit Congestion Notification.. 43

4.5 DECbit.. 45

4.6 Drop Tail & Drop Front on Full Algorithms 46

4.7 Random Drop Algorithm 47

4.8 Early Random Drop Algorithm... 47

4.9 Random Early Detection Algorithm (RED).. 48

4.10 Weighted Random Early Detection (WRED) 52

4.11 Distributed Weighted Random Early Detection (DWRED) 54

4.12 Flow-Based Weighted Random Early Detection (Flow-Based WRED) 54

4.13 Flow Random Early Drop Algorithm (FRED)... 55

4.14 Stabilized RED Algorithm (SRED) 56

4.15 Choose & Keep for Responsive Flows, Choose & Kill for Unresponsive Flows (CHOKe) 58

4.16 Comparison and Classifications of Major IP Level Algorithms.................... 60

4.17 Summary on Active Queue Management Mechanisms 61

CHAPTER FIVE - DESCRIPTION OF OUR APPROACH 62

5.1 Orange, Our Proposed Algorithm 62

5.2 Generic M/M/2 Queue Analysis 65

5.3 M/M/2 Queue Analysis with Heterogeneous Servers 67

5.4 M/M/2 Queue Analysis with a Threshold K=1 70

5.5 M/M/2 Queue Analysis with a Threshold K 72

5.6 Two Server Queue One Server Idle Below a Threshold 76

CHAPTER SIX - MATERIALS AND METHODS 79

6.1 Constructing the Simulation Environment 79

6.1.1 Introducing NS (Network Simulator) 79

6.1.2 Post Simulation Analysis 87

6.1.3 Integrating ORANGE to NS 88

6.2 Topology Alpha with Poisson Sources 91

6.2.1 Simulation of a M/M/1/K Queue 92

6.2.2 Effect of Orange on Simulation's Performance 95

6.3 Topology Bravo with Responsive Sources 97

6.4 Topology Charlie and More on Testing the Download Performance 99

6.5 Topology Delta for Orange's Performance Tests 101

6.6 Topology Echo and Main Experimental Work 102

6.7 Analysis of the Simulation Results 109

6.8 Empirical Validation of Orange's System Parameters 111

CHAPTER SEVEN - CONCLUSIONS **116**

7.1 Drawbacks of Current Active Queue Management Algorithms 116

7.2 Advantages of Orange 117

7.3 Concluding Remarks 119

7.4 Recommendations for future research 120

REFERENCES **121**

APPENDICES **125**

A. Full Source Code of Orange Algorithm: orange.cc & orange.h 125

B. Source Code of the Function: Drop Early Orange 128

C. Source Code of the Function: enqueue 128

D. Queue Size Script: queueSize.awk 130

E. Simulation Statistics Script: avgStats.awk 131

F. Instant Throughput Script: instantThroughput.awk 133

G. Script for Topology Alpha 134

H. Script for Topology Bravo 136

I. Script for Topology Charlie 137

J. Script for Topology Delta 139

K. Script for Topology Echo 142

CHAPTER ONE
INTRODUCTION

1.1 General

The Internet (or simply the Net) is a global information system of interconnected computer networks. It is a network of networks in which users at any one computer can get information according to their access permission from any other computer that is linked by copper wires, fiber-optic cables, wireless connections, and other technologies. It is not only the underlying communications technology, but also higher-level protocols and end-user applications, the associated data structures and the means by which the information may be processed, manifested, or otherwise used. Physically, the Internet uses a portion of the total resources of the currently existing public telecommunication networks. Internet also uses the standardized Internet Protocol Suite (TCP/IP) to serve billions of users worldwide. It is a network of networks that consists of millions of private and public, academic, business, and government networks.

Today, the Internet is a public, cooperative, and self-sustaining facility accessible to hundreds of millions of people worldwide and supports popular services such as most notably the inter-linked hypertext documents of the World Wide Web (WWW), the infrastructure to support electronic mail, online chat, file transfer and file sharing, gaming, e-commerce, social networking, publishing, video on demand, teleconferencing, telecommunications, voice over IP applications.

The origins of the Internet reach back to the 1960s when the United States funded research projects of its military agencies. The original aim was to build robust, fault-tolerant and distributed computer networks. It was foreseen by the Advanced Research Projects Agency (ARPA) of the U.S. government in 1969 and was first known as the ARPANET. The main advantage of ARPANet's design can be explained like that the network could keep functioning even if some parts of it were destroyed because of any attack or other disaster. It was designed by giving

possibility to the messages that they could be routed or re-routed in more than one direction.

1.2 The Evolving Internet

The Internet revolutionizes our society, our economy and our technological systems. No one knows how far, or in what direction, the Internet will evolve. In addition, no one should underestimate its importance.

Since the beginning of networking technology, the number of the host computer systems has increased from four to an estimated 600 million hosts today (Figure 1.1) (www.isc.org). During the last decade, internet continues to grow vigorously, approximately doubling in each year. This exponential grow rate is expected to be continued for the next decades. Future networking demands will require the internet to grow faster. In the future, it is expected that many of new electronic devices will be internet connected; this will require the internet to continue its rapid scaling well into the future.

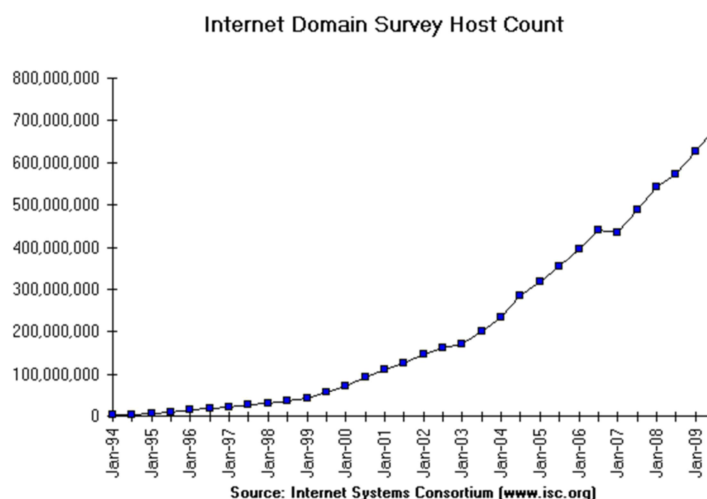

Figure 1.1 Internet domain host count (www.isc.org).

In parallel to the internet growth rate, the need for speed, connectivity, and reliability have become of vital importance. Network performance is vital to businesses operations as well as bringing a product to the consumers through electronic commerce.

As the number of the internet users and the related demand for high-speed networks continue to increase and to be distributed non-uniformly, today's internet backbone has started to operate at its capacity. However, the sufficient infrastructure for high-speed networks is not expected to grow accordingly, due to high investment costs. Because of this trade off, network problems have emerged as a significant problem in all forms of our life, commerce, affecting the way in which many of us work and communicate.

Unfortunately, although its design has focused on robustness, the Internet has the largest performance and availability bottleneck today for end-to-end applications. Congestion causes network connections experience high loss rates during busy hours. Effective congestion avoidance and control policies become essential in order to handle the increasing demand.

The exact causes of Internet performance problems are difficult to be determined because if its scale, heterogeneity, and dynamic nature. However, the design of the Internet protocols had been made in the early 1980s, and it is clear that several of the assumptions have lost their validity today.

> The TCP/IP Internet protocol architecture was designed in the early 1980s, at a time when there were many fewer hosts connected to it and typical links carried only 56 Kbps. Many of the assumptions underlying the Internet's design have changed since then. For example, the designers of Internet congestion control intended it to work well with connections that last many round trips, long enough for end-to-end feedback to work. Most connections today, however, carry only a small number of packets. Transferring a typical 10 Kbyte Web page requires a minimum of six to seven round trips as the server probes the network to determine the maximum rate at which it can send. If there is excess capacity in the network, the overhead of these probes will prevent the server from fully utilizing the

network. If the network is congested, these short, bursty connections will increase the probability of dropped packets. The designers of Internet transport protocols assumed that packet loss rates would be less than 1%, yet current packet loss rates have been measured as averaging 5% to 6%.

Assumptions about Internet routing have changed as well. The Internet was originally designed to provide universal reachability between networks; all network links were available to carry traffic for any host. Today's Internet restricts the exchange of routing information according to business agreements between service providers. These agreements result in situations where A can reach B, and B can reach C, but A cannot reach C. Further, because current Internet routing ignores performance information, two hosts may be forced to communicate over excessively long or overloaded links. Adding a slow link can actually hurt performance, because packets can be routed over it in preference to faster links.

Finally, the Internet was built by a small community of researchers. In that environment, it was reasonable to assume that end hosts would cooperate in the management of network resources. As the Internet has evolved from a research project into a popular consumer technology, this assumption has lost some of its validity. For example, there are several commercial Internet "accelerators" that provide better performance for a single user at the expense of other users. Expecting billions of Internet devices to cooperate to prevent network congestion in the future is arguably too optimistic (Savage *et al.,* 1999, s. 51).

1.3 The Area of Research

In the recent years, the unpredictable growth and the corresponding evolution of the Internet has moreover pointed out the congestion problem, one of the problems that historically have affected the network performance. The network congestion phenomenon is induced when the amount of data injected in the network is larger than the amount of the data that can be delivered to destination.

In many situations in computer communications and networks, there is competition among a collection of competing users for the available resources. These competitions cause congested network traffic, which is undesirable. The competitors are usually frames or packets, of varying sizes, which arrive at unpredictable moments and compete for access to a transmission channel. The resources being shared include the bandwidth of the links, the buffer memory on the routers (switches) where packets are queued waiting to be transmitted over these links, and the processor speeds of these routers. When enough packets are contending for the same link, the queue overflows and packets have to be dropped. It is at this stage that the network is said to be congested.

In a congested network, the gateways along the route would see occasional traffic that go beyond the capacity limit. There are only two possibilities for the gateways along the route; buffer the packets or drop them. Standard gateways usually try to place the incoming packets in their buffers, which work like a basic FIFO ('First In, First Out') queue and only drop packets if the queue is full. Reserving enough buffers for long queues in gateways increases the chance of accommodating short traffic bursts. In spite of high cost of increasing the buffer size in gateways, significant queuing delay problem could not be still avoided by increasing the buffer. Eventually, packet loss will occur regardless of how long the maximum queue is.

The goal of congestion control mechanisms is simply to use the network as efficiently as possible by accomplishing the highest possible throughput, a low packet loss ratio and small delay. Congestion must be avoided because it results in high queue length causing packet delay and loss.

The control of queuing networks has important practical applications in the modeling of manufacturing, telecommunications, and computer systems. In this thesis, we will consider dynamic (state-dependent) control strategies, which can offer significant improvements in network performance over static policies, which do not take into account failures in the network or changes in traffic patterns. For example, by re-routing traffic and re-allocating resources, dynamic routing schemes are capable of responding to the randomly varying demands in a network, managing resources more efficiently and reducing congestion. In particular, we will be

concerned with threshold routing strategies, which are, dynamic routing schemes, which depend on the current state of various queues in relation to fixed threshold values.

1.4 Objectives and Scope

The basic goal of congestion control is to maximize the throughput of the link and minimize the average delay of packets in the network. In addition, it should consider fair allocation of the resources among all the users. More specifically, a congestion control scheme must satisfy:

- Low overhead. In particular, congestion control should not increase traffic during congestion. This is one of the reasons why explicit feedback messages are considered undesirable.

- Responsiveness. The congestion control scheme is required to match the demand dynamically to the available capacity.

- Must continue to work even when the rate of transmission errors, out of sequence packets, deadlocks, and lost packets increases considerable under congestion.

In order to control and avoid congestion, we discuss the problem in terms of congestion control. We propose a new approach, which is implemented in IP level to drop (mark), the packets when the congestion will likely occur. We intend to use an active queue management algorithm in IP level, which we call Orange. Orange will replace RED (Random Early Detection) which will be used at the gateways as the algorithm to decide which packets are to be marked to indicate a congestion condition.

However, the design of an IP level algorithm is not straightforward, because of the heuristic involved with control rules; moreover, the tuning of the parameters of

an algorithm, as scaling factors, membership functions and control rules is a very complex task. Currently there are not many simple methods available for the design of the similar knowledge base.

1.5 Overview of the Thesis

In chapter one, we introduce the subject of the work, namely the congestion and its control. We describe the internet, internet's fast evolution in the last decades, and the result of this evolution, which evolves in congestion.

In chapter two, we define the congestion collapse, which is the undesirable inevitable result of any congested network. We introduce the basic concepts of congestion control including the fairness, the flow control and its difference from congestion control, the classification of congestion control mechanisms. Moreover, we describe the general idea behind the congestion control algorithm of the transmission control protocol (TCP) which is the most commonly used end-to-end, transport layer protocol for today's Internet and multimedia applications that supports flow and congestion control.

In chapter three, as the application of queuing theory provides the theoretical framework for the design and study of computer networks, we revise the basics of queuing theory, which is the mathematical base of our proposed algorithm. We mentioned the general terms including arrival process, service process, queuing discipline and notation of queuing theory as well as the probability theory and the Markov chains, which are used to solve the queuing problems.

In chapter four, we review the literature about the congestion control mechanisms, which have been already studied by several authors. We review the scheduling algorithms, active queue management algorithms including the most widely known type which is RED (Random Early Detection), its derivatives, and performance comparison among them.

In chapter five, we revise the mathematical background behind our proposed algorithm. Generic M/M/2 queue analysis with heterogeneous servers with a threshold is the basis of our proposed algorithm. Markov chains are used for the mathematical analysis. We also describe the details of our proposed algorithm, namely Orange.

In chapter six, we describe the basics of the widely used, public domain discrete event simulator targeted at network protocol research, which we call “NS (Network Simulator)”. We explain the proper method of analyzing the simulation results with “Awk” which is the one of the most interesting text processing languages used for NS trace analysis. In addition, we update NS core to implement our proposed algorithm, which we call “Orange”. Moreover, we give information about the simulation topology and related experimental work to simulate our proposed algorithm. We discuss the results we achieve at the end of the work, advantages of our proposed algorithm, and comparison of our algorithm with similar works.

In chapter seven, we conclude with a summary and identification of key contributions and main findings of this thesis and address the possible avenues of further research based on this work.

CHAPTER TWO
BASICS OF CONGESTION CONTROL

2.1 Overview

A network is considered congested when too many packets try to access the same transmission line, router and other resources. In this case, demanded load exceeds the capacity of network and packets start to be dropped. Additionally, congestion collapse is a condition, which a network can reach, when little or no useful communication is happening due to congestion.

Congestion should be immediately controlled otherwise; there may be many chances of occurring congestion collapse. During congestion collapse, only a fraction of the existing bandwidth is utilized by traffic useful for the receiver. Traffic demand is high but little useful throughput, which is called goodput, is available, and there are high levels of packet delay and loss (caused by routers discarding packets because their output queues are too full). Actions need to be taken by both the transmission protocols and the network routers in order to avoid a congestion collapse and furthermore to ensure network stability, throughput efficiency and fair resource allocation to network users.

2.2 Congestion Collapse

The current congestion control mechanisms for the Internet date back to the 1980's. Those mechanisms were designed to stop congestion collapse for the traffic of 1980's where there was no end-to-end congestion control mechanism in TCP/IP. The Internet first experienced a problem called congestion collapse in the 1980s.

John Nagle identified congestion collapse as a possible problem as far back as 1984 (Nagle, 1984). It was first observed on the early Internet in October 1986, when

the NSFnet phase-I backbone dropped three orders of magnitude from its capacity of 32 kbps to 40 bps, and continued to occur until end nodes started implementing Van Jacobson's congestion control between 1987 and 1988. Congestion collapse is described as a stable condition of degraded performance that stems from unnecessary packet retransmissions (Nagle, 1984). Nowadays, it is, however, more common to refer to “congestion collapse” when a condition occurs where increasing sender rates reduces the total throughput of a network. The existence of such a condition was already acknowledged in Gerla & Kleinrock (1980) that uses the word “collapse”.

We consider a network where sources send at a rate limited only by the source capabilities. Such a network may suffer of congestion collapse, which we explain now on an example.

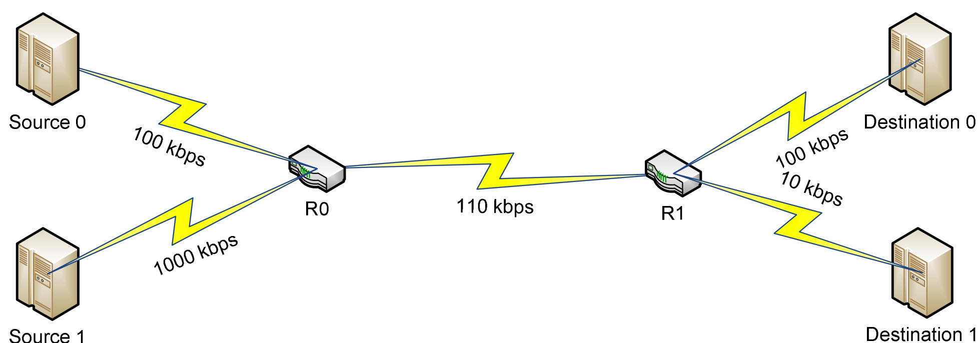

Figure 2.1 A sample network topology to illustrate the inefficiency for unresponsive sources.

Consider first the network illustrated in Figure 2.1, which shows two service providers with two customers each. They are interconnected with a 110 kbps link and do not know each other’s network configuration. Source 0 sends data to Destination 0, while Source 1 sends data to Destination 1, respectively. The sources are limited to send only by their access rates (their first link). Moreover, there are no congestion control feedbacks in the network. There are five links with capacities shown in the Figure 2.1. Source 0 sends at 100 kbps and Destination 0 receives at 100 kbps, while Source 1 sends at 1000 kbps and Destination 1 receives at only 10kbps. Source 0 can send only at 10 kbps because it is competing with Source 1 on the bottleneck link, which sends at a high rate on that link. However, Destination 1 is limited to receive

at 10 kbps. As the Source 1 is unaware of the global network situation, it keeps sending at 1000 kbps (10 times more than the Source 0 on the same bottleneck link). This situation results in the bottleneck link carries 10 times more packets of Source 1 than that of Source 0. Most of the packets from Source 1 will be dropped due to the lack of capacity of the receiver's link. Source 1 will take unnecessarily more bandwidth than Source 0 in bottleneck link resulting in the total throughput of the link will be 20 kbps, which is undesirable.

If Source 1 would be aware of the global situation, and if it would cooperate, then it would send at 10 kbps only on the bottleneck link. In this case, Source 1 would allow Source 0 to send at 100 kbps. The total throughput of the network would then become 110 kbps, which is the ideal case and desirable.

The first example has shown some inefficiency. In complex network scenarios, this may lead to a form of instability known as congestion collapse. This means that the limit of the achieved throughput approaches to zero when the offered load increases.

In the original scenario, throughput is limited by the receiver's link rates, which is 20 kbps. If the sources would cooperate, the throughput would go up to 110 kbps (its maximum rate, which is constrained to this, limit by the bottleneck link). If Source 1 knew that it would never attain more throughput than 10 kbps and would therefore refrain from increasing the rate beyond this point, Source 0 could send at its limit of 100 kbps.

Generally we can say that, as the rate approaches the capacity limit, the throughput curve becomes smoother (this is called the knee), and beyond a certain point, it suddenly drops (this is called cliff) and then decreases further even to zero.

The explanation for this strange phenomenon is congestion. Since both sources keep increasing their rates no matter what the capacities beyond their access links are, there will be congestion in the network. The bottleneck link's queues will grow having more packets from Source 1. Roughly speaking, for every packet from Source 0, there are 10 packets from Source 1. This means that the packets from Source 1

unnecessarily occupy bandwidth of the bottleneck link that could be used by the data flow. The more the Source 1 sends, the greater the congestion problem.

Congestion control deals with such problems. In Ramakrishnan & Jain (1988), the term “congestion control” is distinguished from the term “congestion avoidance” via its operational range: schemes that allow the network to operate at the knee are called congestion avoidance schemes, whereas congestion control just tries to keep the network to the left of the cliff. In practice, it is hard to differentiate mechanisms like this as they all share the common goal of maximizing network throughput while keeping queues short.

The previous discussion has illustrated the “Efficiency Criterion”. In a packet network, sources should limit their sending rate by taking into consideration the state of the network. Ignoring this may put the network into congestion collapse. One objective of congestion control is to avoid such inefficiencies. Congestion collapse occurs when some resources are consumed by traffic that will be later discarded.

2.3 Fairness

Fairness is described as allocating the same share of all available resources among the competing users in a network. We consider the network topology in Figure 2.2. We want to maximize the network throughput in this topology. Sources send at a rate “x_i, $i = 0, 1 ..., I$”, and all links have a capacity equal to “c”. We assume that we implement some form of congestion control and that there are negligible losses. Thus, the flow on “link i” is “$n_0x_0 + n_ix_i$”. For a given value of “n_0” and “x_0”, maximizing the throughput requires that “$n_ix_i = c - n_0x_0$” for “$i = 1,..., I$”. The total throughput, measured at the network output, is thus “$Ic - (I - 1) n_0x_0$”; it is maximum for “$x_0 = 0$”.

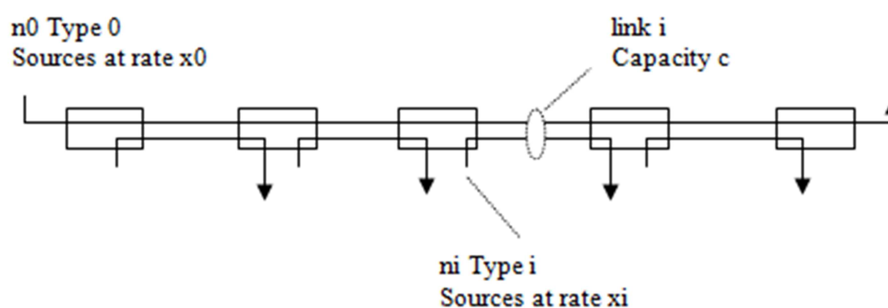

Figure 2.2 A sample network topology to illustrate the fairness.

This example shows that maximizing network throughput as a primary objective may lead to gross unfairness; in the worst case, some sources may get a zero throughput, which is probably considered unfair by these sources. In summary, the main objective of congestion control is to provide both high throughput (efficiency) and some form of fairness.

2.4 Flow Control

Congestion control could be considered to be in networks where neither the sender nor the receiver is involved if the intermediate nodes can take part as controllers and measuring points at the same time. However, most network technologies are designed to operate in a wide range of environment conditions. Consider a network where a sender and a receiver are interconnected via a single link. There are no intermediate nodes in this topology, and thus, no possibility for congestion. Although the congestion phenomenon is not a problem in this topology, the receiver should slow down the sender if it is not fast enough to handle the incoming packets. In this case, the function of informing the sender to reduce its rate is normally called flow control.

“The goal of flow control is to protect the receiver from overload, whereas the goal of congestion control is to protect the network” (Rusmin *et al.,* 2007).Whatever the reason, the underlying mechanism behind congestion control and flow control is

very similar. Feedback messages are used to tune the rate of a flow. Since it is important to protect both receiver and the network from overload at the same time, the sender should send at a rate, which is the minimum of the results of the flow control and congestion control calculations. Because of this similarity, the terms flow control and congestion control are mostly used synonymously. Sometimes flow control is considered as a special case of the congestion control.

2.5 Additive Increase Multiplicative Decrease

Additive Increase Multiplicative Decrease (AIMD) (Dah-Ming & Jain, 1989) algorithm is a feedback control algorithm of TCP's congestion avoidance schema for sharing the available resource among competing users. AIMD algorithm tries to keep the congestion window growing linearly as long as there is no congestion indication (as a congestion indicator, a loss event is generally described to be either a timeout or the event of receiving three duplicate ACKs) in the network. Flows from each source probe for its share of the available resources (i.e. bandwidth) by linearly increasing their transmission rate (window size) until loss occurs (the additive increase stage). When congestion occurs, the sources cut their transmission rates (congestion window) in half in a multiplicative fashion (the multiplicative decrease stage). The result is a saw-tooth behavior that represents the probe for bandwidth. The other forms of AIMD in congestion control are additive increase additive decrease (AIAD), multiplicative increase additive decrease (MIAD) and multiplicative increase multiplicative decrease (MIMD). With these modifications, the AIMD algorithm has been the dominant algorithm in congestion control since the beginning of the congestion control phenomena.

2.6 Overview of TCP's Congestion Control

The transmission control protocol (TCP) (Postel, 1981) is the most commonly used transport layer protocol for today's Internet and multimedia applications. A large amount of Internet traffic is carried by TCP. The Transmission Control Protocol is a reliable, connection-oriented, full duplex, byte-stream, transport layer protocol. In other words, TCP is an end-to-end protocol that supports flow and congestion control.

The congestion control within the TCP plays a critical role in adjusting data sending rate to avoid congestion from happening. Senders to infer network conditions between sender and receiver use acknowledgments for data sent, or lack of acknowledgments. Together with timers, TCP senders and receivers can control the congestion control behavior of a data flow.

TCP implements a window based flow control mechanism. Roughly speaking, a window based protocol means that current window size defines a strict upper bound on the amount of unacknowledged data that can be in transit between a given sender receiver pair. TCP sources waits for an ACK from receiver as a signal to insert a new packet into network without adding to the level of congestion. TCP is said to be self-clocking. In this approach, sources which are responsive (adaptive, or compliant) are considered to reduce their transmission rate. Non-compliant flows can obtain larger bandwidth against the responsive flows.

TCP uses timeouts and duplicate acknowledgements as congestion notifications. Each packet is associated with a timer. If it expires, timeout occurs, and the packet is retransmitted. The value of the timer, denoted by RTO, should ideally be of the order of an RTT. RTT is measured by the TCP connection. If a packet has been lost, the receiver keeps sending acknowledgements but does not modify the sequence number field in the ACK packets. When the sender observes several ACKs acknowledging the same packet, it concludes that a packet has been lost.

The TCP uses a network congestion avoidance algorithm that includes various aspects of an additive-increase-multiplicative-decrease (AIMD) scheme, with other schemes such as slow-start in order to achieve congestion avoidance. Two such variations are those offered by TCP Tahoe and Reno. Before going further about TCP Tahoe and Reno, it is better to remember a short history of evaluation of TCP's Congestion Control Schema.

In 1974, Cerf & Kahn conducted research on packet network interconnection protocols and co-designed the DoD TCP/IP protocol suite. Then, three-way handshake mechanism was described by Tomlinson (1975). In 1981, TCP & IP protocol was first explained in RFC 793 & 791 and it was supported by BSD Unix 4.2 in 1983. In 1984, Nagle's algorithm (Nagle, 1984) was used to reduce the overhead of small packets to predict congestion collapse. In 1986, congestion collapse was first observed. In 1987, Karn's algorithm was used to better estimate round-trip time. In 1988, Van Jacobson's algorithms were described slow start, congestion avoidance, fast retransmit (all implemented in 4.3BSD Tahoe) SIGCOMM 88. The TCP Tahoe and Reno algorithms were retrospectively named after the 4.3BSD Unix operating system in which each first appeared. In 1990, 4.3BSD Reno included fast recovery, delayed ACK's. Improvements were made in 4.3BSD-Reno and subsequently released to the public as "Networking Release 2" and later 4.4BSD-Lite. In 1993, TCP Vegas (not implemented) was described by Brakmo *et al.*, (1993) as a real congestion avoidance schema. In 1994, Explicit Congestion Notification (ECN) was described by Floyd (1994). After that some modifications were followed on TCP's congestion control algorithm including T/TCP Transaction TCP (Braden, 1996), NewReno and SACK TCP Selective Ack (Mahdavi *et al.*, 1996), FACK TCP Forward Ack extension to SACK (Mathis & Mahdavi, 1996). In 2001, Ramakrishnan *et al.*, (2001) added explicit congestion notification bit to the IP headers. In 2004, New Reno modification added to the TCP's fast Recovery Algorithm by Floyd *et al.*, (2004). In 2010, Kuzmanovic *et al.*, (2010) added explicit congestion notification (ECN) capability to TCP's SYN/ACK packets. Floyd *et al.*, (2010) added acknowledgement congestion control to TCP in 2010.

2.6.1 TCP Tahoe and Reno

In order to avoid congestion collapse, TCP uses its own congestion control strategy and for each connection, TCP keeps a congestion window, limiting the total number of unacknowledged packets that may be in transit end-to-end.

The congestion window can be thought of as being a counterpart to advertised window. Whereas advertised window is used to prevent the sender from overrunning the resources of the receiver, the purpose of congestion window is to prevent the sender from sending more data than the network can handle in the current load conditions.

TCP uses slow start mechanism to increase the congestion window after a timeout and after a connection is initialized. In this strategy, the rate of increase is very rapid but the initial rate is slow. Basically, slow start works by increasing the congestion window by one maximum segment size MSS each time for every packet acknowledged so that the congestion window effectively doubles for every round trip time (RTT). It starts with a window of two times the maximum segment size (MSS). Once a loss event has occurred where the initial slow start threshold "ssthresh" is large or the threshold "sstresh" has been reached, the algorithm enters congestion avoidance state. The threshold is updated at the end of each slow start, and will often affect subsequent slow starts triggered by timeouts.

At this point, the connection goes to congestion avoidance phase where the value of congestion window is increased linearly (less aggressively) instead of exponential growth. This linear increase will continue until a packet loss is detected.

Congestion avoidance: As long as non-duplicate ACKs are received, the congestion window is additively increased by one MSS every round trip time. When a packet is lost, the likelihood of duplicate ACKs being received is very high (it's possible though unlikely that the stream just underwent extreme packet reordering, which would also prompt duplicate ACKs). The behavior of Tahoe and Reno differ in how they detect and react to packet loss:

- Tahoe: Loss is detected when a timeout expires before an ACK is received. Tahoe will then reduce congestion window to one MSS, and reset to slow-start state.

- Reno: If three duplicate ACKs are received, Reno reduces the congestion window by half, performs a "fast retransmit", and changes to a state called "Fast Recovery". If an ACK times out, slow start is used as it is with Tahoe.

2.6.2 Fast Retransmit

Duplicate ACKs that were mentioned to be one way of detecting lost packets can also be caused by reordered packets. When receiving one duplicate ACK the sender cannot yet know whether the packet has been lost or just gotten out of order but after receiving several duplicate ACKs it is reasonable to assume that a packet loss has occurred. The purpose of fast retransmit mechanism is to speed up the retransmission process by allowing the sender to retransmit a packet as soon as it has enough evidence that a packet has been lost. This means that instead of waiting for the retransmit timer to expire, the sender can retransmit a packet immediately after receiving three duplicate ACKs.

2.6.3 Fast Recovery

In Tahoe TCP the connection always goes to slow start after a packet loss. However, if the window size is large and packet losses are rare, it would be better for the connection to continue from the congestion avoidance phase, since it will take a while to increase the window size from one to ssthresh. The purpose of the fast recovery algorithm in Reno TCP is to achieve this behavior. In a connection with fast retransmit, the source can use the flow of duplicate ACKs to clock the transmission of packets. When a possibly lost packet is retransmitted, the values of ssthresh and

cwnd will be set to “ssthresh = cwnd/2” and “cwnd = ssthresh” meaning that the connection will continue from the congestion avoidance phase and increases its window size linearly.

In congestion avoidance phase, TCP retransmits the missing packet that was signaled by three duplicate ACKs and waits for an acknowledgment of the entire transmit window to return to the congestion avoidance. If there is no acknowledgment, Reno TCP enters the slow-start state after an experienced timeout. Both of the two algorithms reduce congestion window to one maximum segment size (MSS) on a timeout event.

2.6.4 TCP Vegas

Until Larry Peterson and Lawrence Brakmo, University of Arizona researchers, introduced TCP Vegas in mid 1990s, where timeouts were set and round-trip were measured for every packet in the transmit buffer, all TCPs setting timeouts and measuring round-trip delays were based upon only the last transmitted packet in the transmit buffer. In addition, additive increases are used in the congestion window by TCP Vegas.

2.6.5 TCP New Reno

The difference between the TCP Reno and the TCP New Reno is the improved retransmission during the fast recovery phase. During fast recovery, a new unsent packet from the end of the congestion window is sent for every duplicate ACK that is returned to TCP Reno, to keep the transmit window full. The sender assumes that the ACK points to a new hole for every ACK that makes partial progress in the sequence space and the next packet beyond the acknowledged sequence number is sent.

New Reno has the capability of filling large holes or multiple holes in the sequence space - much like TCP SACK. It gets this capability from the timeout timer which is reset whenever there is progress in the transmit buffer. During the hole filling process in New Reno, high throughput is maintained because it can send new packets at the end of the congestion window during fast recovery; even there exist multiple holes, of multiple packets each. TCP records the highest outstanding unacknowledged packet sequence number when it enters fast recovery. It returns to the congestion avoidance state when this sequence number is acknowledged.

When there are no packet losses but instead they are reordered by more than three packet sequence numbers, a problem occurs with New Reno. When this kind of conditions occurs, it enters fast recovery mistakenly. After the delivery of reordered packet, ACK sequence-number progress occurs. To the end of fast recovery, every bit of sequence-number progress produces a duplicate and retransmission that is immediately acknowledged which is needless.

The aim of TCP Congestion control scheme is to decrease the delays and increase the throughput. It introduces the concept of fairness and tries to avoid congestion collapse. Because more than 95% of today’s flows are TCP flows, this kind of congestion control scheme makes the internet more stable and robust.

2.7 Classification of Congestion Control

Congestion control is a mechanism to inform the sender about the changing condition of the network. There are two basic methods available for congestion control; rate based and window based.

In rate-based control, sources know an explicit rate at which they can send (a specific data rate). The rate is assigned to the source at the negotiation phase of a connection (ATM or RSVP cases), and the receiver or a router informs the sender of a new rate if the network’s state changes at later stages (ABR class of ATM).

In window-based control, the sender maintains a special window (a predetermined number of packets or bytes that it is allowed to be sent before any feedback arrives from the network or receiver). In other words, congestion window is a limit on the number of packets that the sender is able to send. The sender increases the window size as long as it gets positive feedbacks (acknowledgements) from the receiver. The sender decreases the rate at which it sends in case of a packet failure. Since the sender's behavior is controlled by the presence or absence of incoming feedback from the network, window-based control is said to be self-clocking.

There are three possibilities available for a packet in a network. Packets can be delayed, dropped, or changed. Packets can be delayed due to the distance, queuing in the nodes, or retransmissions at the link layer. Packets can be dropped because buffer memories in the nodes could be full, packets could not be admitted (quality of service applications), or the routers could be malfunctioning. Packets can be changed, because the link noises could make packet be changed. All of these reasons indicate congestion in the network.

There are two different approaches available in window-based control; hop-by-hop and end to end. In hop-by-hop approach, sources need feedback from the next hop in order to send any amount of packets. The next hop obtains some feedback from the following hop and so on. The feedback may be positive (credits) or negative (backpressure). In the simplest form, the protocol is stop and go. Hop by hop control is used with full duplex Ethernet using 802.3x frames called "Pause" frames.

In end-to-end approach, sources continuously obtain feedback from all nodes it uses. The feedback is piggybacked in packets returning towards the source, or it may simply be the detection of a missing packet. Sources respond to negative feedback by reducing their rate and to positive feedback by increasing it. All reactions to feedback are left to the sources in end-to-end control whereas the intermediate nodes take action for the feedback in hop-by-hop control.

Rate-based control is easy to implement, and more proper for streaming media applications because it does not stop if no feedback arrives. These types of applications should keep sending their packets regardless of the feedback from the

network. If window-based control is used, re-ordering and delays of the packets make the streaming application meaningless or hard to understand.

From a network perspective, window-based flow control is more proper because the sender will automatically stop sending when there is an incipient congestion indication in the network. The disadvantage of window-based control is that it may lead to traffic bursts.

Sender sends the packets in a regular spacing. If the network is congested, then these packets must be queued at the bottleneck queue. As soon as the congestion is resolved, the bottleneck queue starts to send the corresponding queued packets with a reduced spacing (depending on the capacity of the remaining part of the link). This effect (pacing effect) also occurs when the acknowledgements (and not the data packets) experience congestion.

In addition to the effect of congestion, if the window is too small, the link will be underutilized. In order to utilize the link, the sender must be able to increase its rate as long as the link's capacity. Increasing the window by one packet in response to an ACK is not enough. Increasing the rate means to have the window grow by more than one packet per ACK, and decreasing it means reducing the window size. The ideal window size (which has the sender saturate the link) in bytes is the product of the bottleneck capacity and the RTT. Thus, in addition to the necessity of precise RTT estimation for the sake of self-clocking (i.e. adherence to the conservation of packets principle), the RTT can also be valuable for determining the ideal maximum window.

2.8 RTT Estimation

The Round Trip Time (RTT) is defined as the time between sending a packet into the network and receiving back the corresponding ACK for that packet. The RTT is an important parameter of various algorithms in congestion control. In end-to-end congestion control schemas, sources retransmit their packets, which have been lost

on the network because of an incipient congestion for reliable transmission. Sources use acknowledgement mechanism for their packets, which have a special consecutive number. If any of them is missing for a long time, the sources assume that the packet has been dropped. This mechanism is called Automatic Repeat Request (ARQ), requires a timer value that is initialized with a certain timeout value when each packet is sent.

Finding the right timeout value is an important subject in the context of congestion control. Larger values of this timer can cause longer times for a packet to be retransmitted. This situation will negatively affect the delays and the throughput in the network, because sources reduce their rates unnecessarily. Smaller values of this timer can cause a packet to be retransmitted unnecessarily. Therefore, network capacity will be wasted. If we omit the transient delay changes in the network, the ideal value for a timeout is said to be generally one RTT or a function of an RTT.

Predetermined value of timeouts may result in performance issues because of the state changes within the network (delay in queues, path changes and so on). Timeouts values must be adaptive over the history of RTT samples.

As a common rule of thumb, RTT prediction should be conservative: generally, it can be said that overestimating the RTT causes less harm than underestimating it. An RTT estimator should be robust against short dips while ensuring appropriate reaction to significant peaks.

CHAPTER THREE
BASICS OF QUEUEING SYSTEMS

3.1 General

Queuing is an aspect of our modern life that we may encounter at every step in our daily activities. The queuing arises whenever a shared facility needs to be accessed for service by a large number of jobs, customers or data packets. Our interest of queuing systems arises for its relation to its use in the study of communication systems and computer networks. The various computers, routers and switches in such a network may be modeled as individual queues with respect to their buffer memory coupled with service elements. The whole system may itself be modeled as a queuing network providing the required service to the messages, packets or cells that need to be carried. Application of queuing theory provides the theoretical framework for the design and study of such networks. Throughout our thesis, we are going to use the theoretical background and notation of queuing systems to analyze our proposed algorithm.

The objective of queuing theory is to understand such queuing phenomenon in order to predict the performance, control, and sometimes optimize the system where the queuing occurs. Due to the range of applicability and potential gain of controlling these systems, proper understanding of queuing can be a powerful tool.

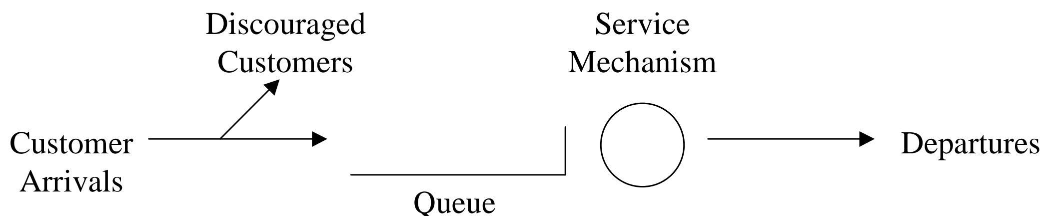

Figure 3.1 General queuing system.

In general, a queuing system involves customers who enter the system, wait in line (a queue), are served, and leave the system as shown in Figure 3.1. The key features of queuing systems can be classified as characteristics of arrivals, service discipline, and characteristics of service.

3.2 The Arrival Process and the Queue

The queue is characterized by the maximum permissible number of customers that it can contain. This number is either potentially infinite or finite. It is dependent on the physical limitations of the memory "available space" of the system. The ease with which we can analytically modeling a queuing system of unlimited length is much greater than that with which we can model a limited queue situation. We will further use infinite capacity of modeling a queue in our thesis.

The arrival process is characterized by the arrival rate (λ). The arrival time is simply the amount of time between two adjacent frames. "Arrival rate" is the reciprocal of arrival time (1/λ).

The arrival process has three main characteristics:

- The size of the population. Most queues arise from a population that is very large compared to the overall queue size.
- The pattern of the arrival process. Most frames join the queue in a random nature with each one being independent of the others, both in their chance of joining the queue and in the time in which they join.
- The behavior of the arrivals. Most people once they have joined the queue remain in it known as "settling".

Some, however, refuse to join because they feel it is too long known as "balking". Others once in, leave before they reach the service, as they become inpatient known as "reneging". We will further use infinite population, exponentially distributed inter-arrival times and settling behavior of modeling a queue in our thesis.

3.3 The Service Process and the Server

Systems are usually described in terms of the number of channels they have and the number of phases they have. The channels are the number of areas providing the service known as “server”.

The service process is characterized by the service time (μ). The service time is simply the amount of time required to transmit a frame. Since the bit-rate of the channel is constant, this is strictly proportional to the length of the frame. “Service rate” is the reciprocal of service time (1/μ). In some types of services, the time taken to see each patient is constant, but in many the time taken to see the patient is variable and in most systems, these are random and can be described by the negative exponential distribution. In simple terms, this states that the probability of a very long service time is low, with most people being seen around the average service time.

3.4 Queuing Discipline

Queuing discipline refers to the rule by which customers in the queue receive their service.

- First in first out (FIFO). This is the approach to handling data packet requests from queues or stacks so that the oldest request is serviced.
- Shortest service time (SST). This is where the patient with the shortest procedure is seen first. It is seen in the selection of some types of procedures for operating lists.
- Last come first served (LCFS). The obvious example here is people getting out of a lift, those who entered last get out first.

- Earliest due date (EDD). This may occur when the latest date for treatment has been fixed. For instance, when patients approach as the maximum period they are allowed to wait.

- Shortest weighted service time (SWST). This is similar to SST, but can be weighted according to agreed criteria of how important it is to see that particular patient. To be successful the weights should not be arbitrary, but should be tied to defined criteria.

3.5 Probability Distribution of Arrival or Service Times

The statistical pattern by which the customers arrive at the queuing system occurs either according to some predetermined schedule or at random. If the pattern is scheduled, then analytical model is unnecessary. If the pattern is random, then it is necessary to determine the specific type of probability distribution of the time between consecutive arrivals to the queue or departures from the servers.

3.5.1 Poisson Distribution

Arrivals to the queuing system or departures from servers occur randomly, but a certain average rate. An equivalent assumption is that the probability distribution of the time between consecutives arrivals is exponential, and that the number of arrivals during a certain time interval is independent of the number of arrivals that have occurred in previous time intervals (i.e. "memory-less") (see Figure 3.2). The mathematical relationship of the Poisson distribution is;

$$P_n(t) = \frac{(\lambda t)^n e^{-\lambda t}}{n!} \qquad \text{Eqn 3.1}$$

where;

$P_n(t)$ = probability that there will be exactly n customers into the system during a specified time increment, t.

λ = mean arrival time.

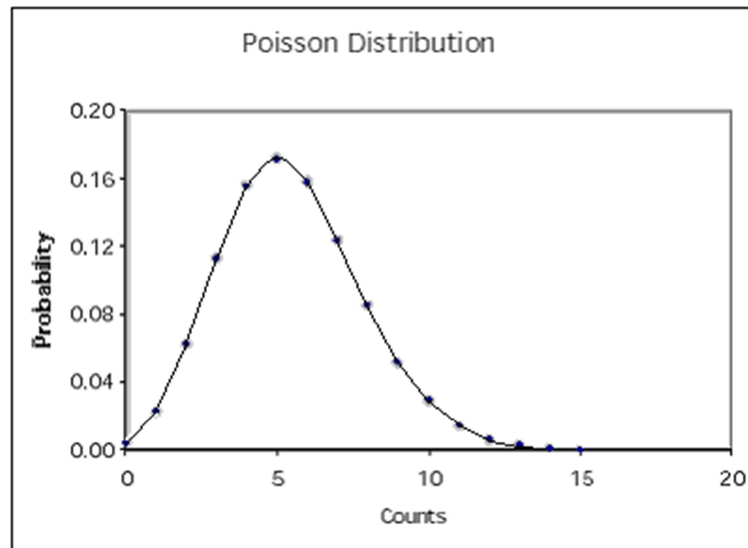

Figure 3.2 Poisson distribution.

Although the Poisson distribution represents the arrival pattern for many queuing systems, it does not portray the situation for all settings. It is crucial, therefore, to verify the specific type of arrival pattern for the system under investigation prior to the selection of the analytical model.

3.5.2 Exponential Distribution

The probability of completing a service to a customer in any subsequent time interval is independent of how much service time has already elapsed for that customer. The exponential probability distribution (see Figure 3.3) has a memory-less property and is given by the following formula;

$$P(t > T) = e^{-\mu t} \qquad \text{Eqn 3.2}$$

where;

$P(t > T)$ = the probability that the service time "t", exceeds a specific time "T" for a mean service rate of "μ".

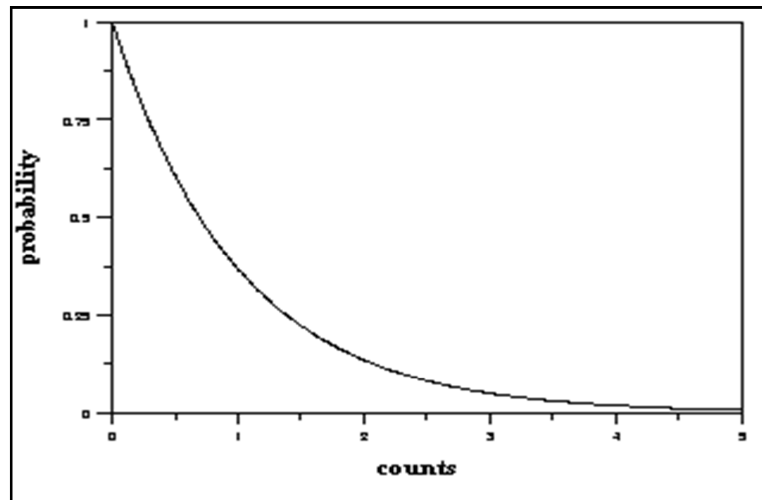

Figure 3.3 Exponential distribution.

3.5.3 Gamma (Erlang) Distribution

The gamma distribution has two parameters and thus can represent an entire family of distributions. The ability to vary these parameters easily gives the Erlang distribution great flexibility in modeling service situations that are characterized by a number of subtasks. The Erlang distribution is of particular value when the type of service to be provided a customer consists of "k" subtasks, each of which has an identical exponential distribution. In reality, however, a task needs only to behave in total as though it were the sum of "k" identically distributed tasks; it does not have to be capable of actual subdivision. The mean service time of each of the "k" subtasks would then be "1/kμ". The mean of the total service time is then "k/kμ" or "1/μ". This value represents the expected completion time of the entire task. The Erlang probability distribution of the total service time "t" is

$$f(t) = \frac{t^{k-1} e^{k\mu t} (k\mu)^k}{(k-1)!} \quad \text{Eqn 3.3}$$

Notice that, for the case when “k = 1”, the Erlang distribution becomes the exponential distribution. Also, if “k = ∞”, the service time will become a constant. See Figure 3.4.

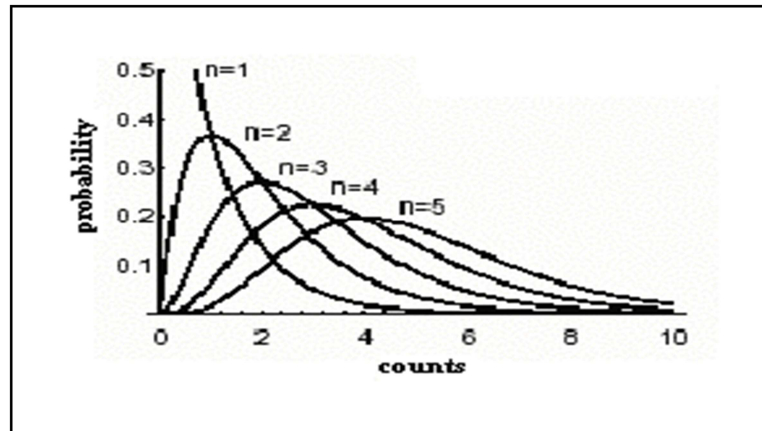

Figure 3.4 Gamma (Erlang) distribution.

3.6 Notation for Queuing Systems

As we describe, a queue is described as follows:

- Arrival process of requests;
- List of requests waiting service;
- Service policy adopted for the different requests in the list;
- Number of servers that characterize the maximum number of simultaneously served requests;
- Statistics of the service duration of each request.

To describe all the above aspects, the following notation has been introduced by Kendall. It has the form “A/B/c/K/m/Z” where “A” describes the type of the arrival process (e.g., “A = M” for a Poisson process; “A = GI” for a renewal arrival process). “B” represents the statistics of service duration of a request (e.g., “B = M” for an exponentially distributed service duration; “B = G” for a generally distributed service process). “c” indicates the number of servers (i.e., “c” can be a suitable

integer value or even infinity). ”K” denotes the number of rooms for service requests in the queuing system, including the currently served request: “K” can be a given finite value or infinity (in this case it is omitted in the notation). “m” specifies how many sources can produce requests of service: “m” can be a given finite value or infinity (in such case it is omitted). Finally, “Z” gives the queue discipline.

Usually the shorter notation “A/B/c” is used and it is assumed that there is no limit to the queue size, the customer source is infinite, and the queue disciple is FIFO.

For A and B the following symbols are traditionally used:

- GI; general independent inter-arrival time,
- G; general service time distribution,
- Hk; k-stage hyper-exponential inter-arrival or service time distribution,
- Ek; Erlang-k inter-arrival or service time distribution,
- M; Exponential (Markovian – memory-less) inter-arrival or service time distribution,
- D; deterministic (constant) inter-arrival or service time distribution.

3.7 Queues and Probability Theory

Probability theory is the basic mathematical tool to analyze algorithms and systems in computer science. In probability theory, a stochastic process or random process is the collections of interdependent random variables. It is the counterpart to a deterministic process (or deterministic system).

Queues are special cases of stochastic processes that are represented by a state X(t) denoting the number of queued “entities”. The queue is characterized by an arrival process of service requests, a waiting list of requests to be processed, a

discipline according to which requests are selected in the queue to be served and a service process. A stochastic process is identified by a different distribution of random variable “X” at different time instants “t”. A stochastic process is characterized by:

- The state space, that is the set of all the possible values that can be assumed by “X(t)”. Such space can be continuous or discrete (in such a case the stochastic process is named chain).
- Time variable: variable “t” can belong to a continuous set or to a discrete one.
- Correlation characteristics among “X(t)” random variables at different instant “t” values.

In order to account for these correlation aspects, we describe “X(t)” in terms of its joint probability distribution function at different instants “$t = \{t_i, t_2, ..., t_n\}$” and for different values “$x = \{x_i, x_2, ..., x_n\}$” for any “n”:

$$PDF_x(x,t) = Prob\{X(t_1) \leq x_l, X(t_2) \leq x_2,...,X(t_n) \leq x_n\} \qquad \text{Eqn 3.4}$$

This process “X(t)” is strict-sense stationary if for any “n” value and “t” the following equality hold (i.e., distribution PDFx(x,t) is invariant to temporal translations):

$$PDF_x(x,t+\tau) = PDF_x(x,t) \qquad \text{Eqn 3.5}$$

Typically, we use the wide-sense stationary requiring that the expected value “E[X(t)]” is independent on “t” and the correlation “E[X(t)X(t+τ)]” is independent on “τ”. A process is independent when for any “n” and “t” we have:

$$PDF_X(x,t) = Prob\{ X(t_1) \leq x_l \} Prob\{ X(t_2) \leq x_2 \}... Prob\{ X(t_n) \leq x_n \} \qquad \text{Eqn 3.6}$$

The same relationship holds in terms of probability density functions (we take partial derivatives on the left side and we take the total derivatives of the single distributions on the right side). In the case of an independent process, the random variables at the different instants are completely uncorrelated.

A special type of stochastic process is a Markov chain, where “X(t)” can only assume discrete values and is characterized by the fact that its state at instant “t_n+i”, “X(t_n+i)”, depends only on the state at the previous instant “t_n, X(t_n)”. The chain evolves in time by making transitions between states. The stochastic process evolution is only characterized by its state value at the present instant, but not on the time already spent in this state. This memory-less characteristic is guaranteed only by state sojourn times exponentially distributed in the case of a continuous-time chain (whereas the geometric distribution must be considered for a discrete-time chain). The formal definition of a continuous-time Markov chain “X(t)” is:

$$Prob\{X(t_{n+1})=x_{n+1}|X(t_n)=x_n,X(t_{n-1})=x_{n-}\ldots,X(t_1)=x_1\}=Prob\{X(t_{n+1})=x_{n+1}|X(t_n)=x_n \qquad \text{Eqn 3.7}$$

In case that the time instants where the chain can perform transitions are discrete, we have a discrete-time chain. A Markov chain is characterized by means of the mean rates that correspond to the different transitions from a state to another. Some important sub-classes of Markov chains are as follows:

- *Birth-death chains*, where from state “X = i”, it is only possible to go to states “X = i-1” or “X = i+l”.
- *Renewal processes:* these are “point” processes (i.e., arrival processes or only-birth processes) like the arrival of points on the time axis. The inter-time from adjacent points (i.e., arrivals) are independent identically distributed. A special case of renewal processes if the Poisson arrival process, where inter-arrival times are exponentially distributed with a constant rate.

- *Semi-Markov chains:* these are chains where the sojourn time in a state has a general distribution. By observing these chains at the state transition time, we obtain an imbedded Markov chain, which can be considered (and solved) as a discrete-time Markov chain. Semi-Markov chains will be used to solve “M/G/l” (and “G/M/l” queues).

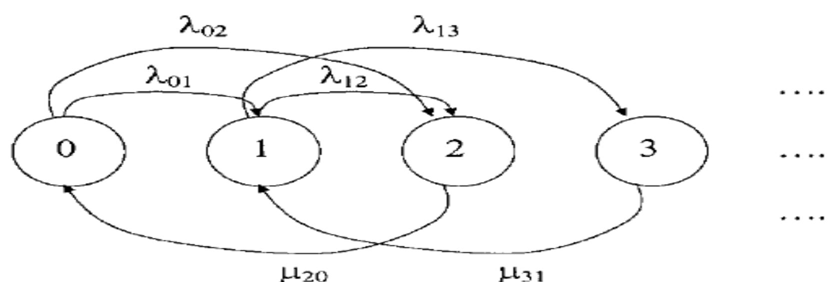

Figure 3.5 Continuous-time Markov chain with denoted mean transition rates.

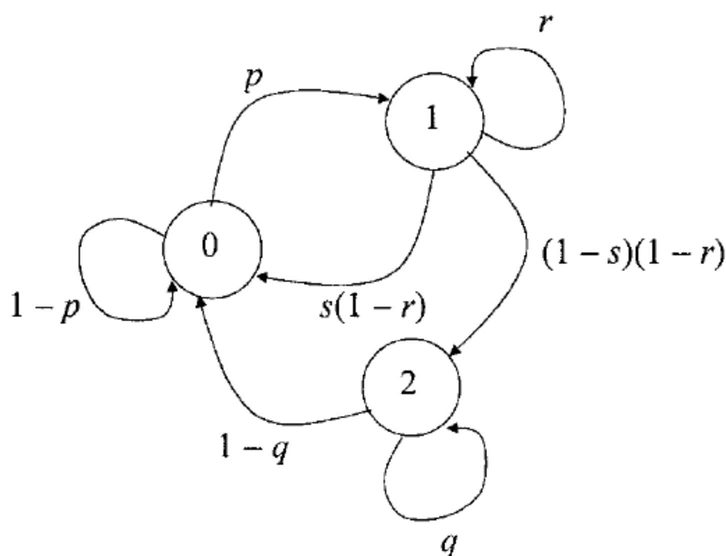

Figure 3.6 Discrete-time Markov chain with denoted transitional probabilities.

Markov chains are characterized by state diagrams that describe the states (denoted by circles) and the allowed transitions (denoted by arrows) among them. In the case of a continuous-time chain, transitions may occur at any instants and are characterized by mean rates of exponential distributions (see Figure 3.5). Whereas, for discrete-time chains, transitions occur at given time instants and are characterized by transition probabilities that characterize the geometric distributions of the state

sojourn times (See Figure 3.6). In this case, states may have transitions into themselves. The sum of all the transitional probabilities leaving a state must be equal to one (normalization condition).

3.8 Birth-death Markov Chains

Queuing systems are generally characterized by continuous-time Markov chains that describe the behavior of a "population" with states representing the natural numbers {0, 1, 2 . . .}. For a generic state "k", only transitions to states "k-1" and "k+1" are allowed. Let us denote:

- λ_i, the mean birth rate from state "i" to state "i+1";
- μ_m, the mean death rate from state "m" to state "m-1";
- P_n, the probability of state "n".

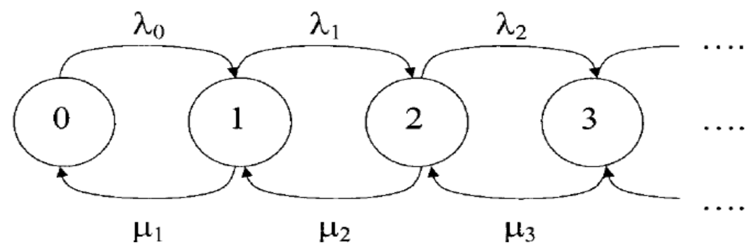

Figure 3.7 Birth-Death Markov chain.

A general example of Markov chain is shown in Figure 3.7. In this figure, we assume an infinite number of states. The time behavior of this chain is described by the Kolmogorov - Chapman equations. A sufficient (equilibrium) condition to a have a steady-state behavior is the following ergodicity condition:

$\exists$ an index k_0 so that for each $k \geq k_0$, we have $\lambda_k < \mu_k$.

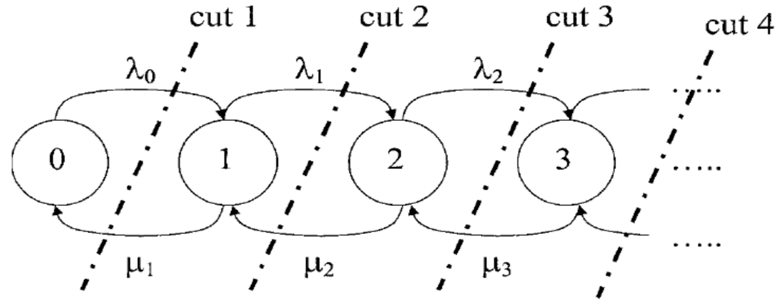

Figure 3.8 Cuts for the balance equations at equilibrium.

This condition expresses the fact that there is a state beyond which the birth rate is lower than the death rate. Assuming that the equilibrium condition is fulfilled, we study the chain in Figure 3.7 at equilibrium by imposing the balance of the "fluxes" across any given closed curve encircling states in the diagram. Many equilibrium conditions can be stated; namely circles around a state or cuts that intercept transition arrows between two states. The simplest approach is to make cuts between any couple of states in the diagram as shown in Figure 3.8 and to write the corresponding balance equations described below:

Cut 1 balance: $\lambda_0 P_0 = \mu_1 P_1 \Rightarrow P_1 = \frac{\lambda_0}{\mu_1} P_0$

Cut 2 balance: $\lambda_1 P_1 = \mu_2 P_2 \Rightarrow P_2 = \frac{\lambda_1}{\mu_2} P_1 = \frac{\lambda_1}{\mu_2}\frac{\lambda_0}{\mu_1} P_0$ Eqn 3.8

....

Cut i balance: $\lambda_{i-1} P_{i-1} = \mu_i P_i \Rightarrow P_i = \frac{\lambda_{i-1}}{\mu_i} P_{i-1} = P_0 \prod_{n=1}^{i} \frac{\lambda_{n-1}}{\mu_n} \quad \forall i \geq 1$

All the state probabilities are expressed as functions of both the transitional rates and the probability of state "0", namely P_0. Therefore, we impose a normalization condition in order to obtain P_0:

$$\sum_{i=0}^{\infty} \mathrm{P_i} = 1 \Rightarrow \mathrm{P_0} \sum_{i=0}^{\infty} \frac{\mathrm{P_i}}{\mathrm{P_0}} = 1 \Rightarrow \mathrm{P_0} \left(1 + \sum_{i=1}^{\infty} \prod_{n=1}^{i} \frac{\lambda_{n-1}}{\mu_n} \right) = 1 \Rightarrow \mathrm{P_0}$$

$$= \frac{1}{1 + \sum_{i=1}^{\infty} \prod_{n=1}^{i} \frac{\lambda_{n-1}}{\mu_n}}$$

Eqn 3.9

Birth death Markov Chains comprises the mathematical model to construct the state probabilities of our algorithm. Therefore, we use the birth death Markov Chains background in our work.

CHAPTER FOUR
LITERATURE REVIEW

4.1 Congestion Avoidance Mechanisms

TCP congestion control mechanism is effective once the network is congested. It does not try to avoid congestion without congestion notifications (triple duplicates and timeouts). TCP probes the network by increasing the window size (packet send rate) until the point at which congestion happens, and then decreases the window size after any congestion notification. TCP needs to lose packets to be aware of the available bandwidth of the connection. Another alternative to congestion control is congestion avoidance. It aims to avoid congestion by predicting the incipient congestion to notify the responsive sources when congestion is about to happen. Responsive sources reduce its packet-sending rate so that they will be aware of the incipient congestion before they lose any packet.

However, congestion avoidance mechanism has still some disadvantages. It considers the responsive flows, which reduce the rate at which they send when it gets a signal from the network about the congestion. It awards unresponsive flows like UDP flows, which have a constant sending rate during connection. In this scheme, the delays in the network will increase because packets will be dropped only after the queues have already built up. Another disadvantage is that, it needs global synchronization. All the flows will reduce their sending rate simultaneously, which will decrease the throughput. Finally, although most of the flows are TCP flows, there are still some flows are not TCP compliant. Those flows may not respond to the congestion and will eventually take over all the links' bandwidth and exhaust the network.

With all these disadvantages, researchers begin to consider the needs of controlling congestion at the gateways (IP Level). Two types of different approaches arise to control congestion; scheduling algorithms and queue management algorithms.

4.2 Scheduling Algorithms

A scheduling algorithm (Figure 4.1) keeps separate queues for each flow. A flow cannot degrade the quality of other flows. The advantage of the scheduling algorithms is to give a fair share of the bandwidth to all competing users. However, it does not scale well to a large number of flows. It requires heavy and expensive computations and more memory resources.

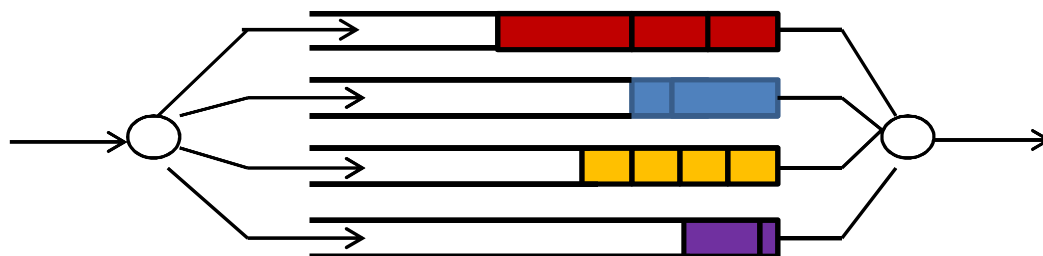

Figure 4.1 Scheduling algorithm.

In the future, new applications such as teleconferencing, voice over IP (VoIP), IPTv will get more usage in networking world. These applications will require the ability of the network to guarantee the demanded bandwidth. They would expect the network to ensure that each flow of traffic receives its fair share of the bandwidth and is able to provide an upper bound on the end-to-end delay. This demand requires a Quality of Service (QoS) mechanism to manage limited resources among competing users. Quality of Service mechanism uses a traffic-scheduling algorithm at the output links of switches and routers on the network. The main function of the packet scheduler at the output link is to determine the next packet for transmission among the packets, which wait for transmission from different flows. A scheduling algorithm must satisfy the following properties; fairness, efficiency, and low latency.

Fairness is described as allocating the same share of all available resources among the competing users in a network. This ensures that, a flow takes it fair share of the available bandwidth even if an unresponsive flow tries to transmit their packets at a rate faster than its fair share.

Latency is generally defined as how much time it takes for a packet of data to get from one designated point to another. In some cases, latency is measured by sending a packet that is returned to the sender and the round-trip time is considered the latency. For the applications that need a guaranteed rate, latency is measured as the length of time it takes a new flow to begin receiving service at the guaranteed rate.

Efficiency is the measurement of performance of packet switched networks. Efficiency is mostly affected by the processor speeds, and hardware resources in a gateway. A packet scheduler should make its scheduling decision in a time as small as possible to achieve a higher efficiency. Hence, it is desirable that the time to enqueue a received packet or to dequeue a packet for transmission is as independent as possible of the number of flows sharing the output link.

Scheduling algorithms are generally classified into two categories: sorted priority schedulers and frame-based schedulers. Sorted priority schedulers keep a global variable called as the virtual time or the system potential function. This variable is used by a sorted priority scheduler to compute the timestamp for each packet indicating the relative priority of the packet for transmission over the output link. Packet scheduler makes a list of pickets' timestamps in an increasing order. Each packet is transferred to their output link according to their timestamps. Most known sorted priority schedulers are Weighted Fair Queuing (WFQ), Self Clocked fair Queuing (SCFQ), Start Time Fair Queuing (SFQ), Frame Based Fair Queuing (FFQ), and Worst Case Fair Weighted Fair Queuing (WF2Q).

The sorted priority schedulers vary as how they can calculate the global virtual time function. There are two different performance criteria behind sorted priority schedulers;

- The complexity of computing the system virtual time.

A per packet work complexity of O(1) is most desirable. For WFQ, the worst case complexity is O(n) where n is the number of flows sharing the same output link. However, in a number of schedulers such as SCFQ, SFQ and FFQ proposed in recent years, the complexity of the computing the virtual time is O(1).

- The complexity of maintaining a sorted list of packets based on their timestamps, and the complexity of computing the maximum or the minimum in this list prior to each packet transmission.

For n flows the work complexity of the scheduler prior to each packet transmission is O(log n).

In frame-based schedulers such as Deficit Round Robin (DRR) and Elastic Round Robin (ERR), the scheduler visits all the non-empty queues in a round robin order. During each service opportunity of a flow, the intent of such a scheduler is to provide to the flow an amount of service proportional to its fair share of the bandwidth. The frame-based schedulers do not maintain a global virtual time function and do not require any sorting among the packets available for transmission. This reduces the implementation complexity of frame based scheduling disciplines to O(1), making them attractive for implementation in routers, and especially so, in hardware switches.

Elastic Round Robin (ERR) is a recently proposed frame based scheduling discipline for best effort traffic, that achieves very good efficiency with a low per packet work complexity of O(1) with respect to the number of flows. In addition, it has better fairness properties than other schedulers of equivalent work complexity such as DRR. ERR can also be easily adapted for scheduling guarantee rate connections, and that it belongs to the class of Latency Rate (LR) Servers, with a latency bound significantly lower than those of other scheduling disciplines of comparable work complexity. These properties of ERR makes it an attractive scheduling discipline for both best effort and guaranteed rate services.

4.3 Active Queue Management Algorithms

Active queue management algorithms use a single FIFO (First In First Out) queue for all flows flowing through the router. It uses a certain algorithm manage the length of the packet queue by dropping packets when necessary or appropriate. This kind of approach requires no state information and scales well.

TCP congestion control algorithm detects congestion only after a packet has been dropped along the path. Increasing the queue size does not solve the congestion problem. The responsive sources detect packet loss as a congestion indicator. If the packets will not be dropped because of high queue sizes at the gateways, sources will keep increasing the sending rate causing longer delays in the network, which is not desirable. It is important to find out the ideal maximum queue length. This parameter should be tuned properly in order to minimize the average delay in the network. In order to maximize the utilization of the link, we consider a single flow and a single link. Links are best characterized by their bandwidth-delay product for end-to-end systems where bandwidth (capacity of the link) in bits per second and delay (average RTT of flows) of the link in seconds. Generally, we can say that the queue limit of a router should be set to the bandwidth-delay product. This general rule is outdated, because it leads to quite a large buffer space at the gateways. As an update to this general rule, dividing the bandwidth-delay product by the square root of the number of flows in the network gives better results.

Queues should be generally kept short. Therefore, it is important to have mechanisms that keep throughput high but average queue sizes low.

Active Queue Management (AQM) is an IP level (gateway based) congestion control scheme where gateways notify the sources of incipient congestion. The aim of AQM systems is to keep the average queue sizes at the gateways low. Keeping the queue sizes low has some advantages including,

- Provide queue space to absorb bursts of packet arrivals,
- Avoid lock-out and bias effects, from a few flows dominating queue space,

- Provide lower delays for interactive applications.

All AQM schemes detect impending queue buildup and notify the sources before the queues at the gateways overflows. AQM algorithms differ in the mechanism used to detect congestion and in the type of control method used to achieve a stable operating point for the queue size. Trying to keep the queue size stable at a desired level causes a tradeoff between link utilization and queuing delay. A short queue reduces latency at the router but setting the target queue size too small may reduce link utilization by limiting the router's ability to buffer short bursts of arriving packets.

The way in which the congestion notification is delivered to the sources is the other important property of AQM schemes, which affects the performance. Two different alternatives are available used to notify the sources.

Early Congestion Notification (ECN) adds an explicit signaling mechanism by allocating bits in the IP and TCP headers of the packets flowing through the router. In turn, the destination will transmit such information to the source piggybacking it into the acknowledgement message. Another way of speaking, gateways signal congestion to the sources by "marking" a packet (setting a bit in the header).

The current Transmission Control Protocol, which is the dominant transport protocol in today's internet, is not able to manage the ECN bit. Generally speaking, gateways drop the packets randomly with a probability when the queue sizes grow up in order to notify the sources about the incipient congestion.

4.4 Explicit Congestion Notification

Explicit Congestion Notification (ECN), which is an extension to the Internet Protocol, is defined in Ramakrishnan et al., (2001). It features end-to-end notification of network congestion without dropping packets. This feature is optional and it is only used when both of the endpoints signal that they want to use it. Dropping packets to signal congestion is the traditional way in TCP/IP networks. After ECN is

negotiated, a router, which is ECN-aware, may set a bit in the IP header instead of dropping a packet in order to signal the beginning of congestion. The receiver side echoes back the congestion indication to the sender side and it reacts as a packet drop were detected.

For networks with mechanisms for the detection of incipient congestion, the use of ECN mechanisms for the notification of congestion to the end nodes prevents unnecessary packet drops. For bulk-data connections, the user is concerned only with the arrival time of the last packet of data, and delays of individual packets are of no concern. For some interactive traffic, however, such as telnet traffic, the user is sensitive to the delay of individual packets. For such low-bandwidth delay-sensitive TCP traffic, unnecessary packet drops and packet retransmissions can result in noticeable and unnecessary delays for the user. For some connections, these delays can be exacerbated by a coarse-granularity TCP timer that delays the source's retransmission of the packet.

A second benefit of ECN mechanisms is that with ECN, sources can be informed of congestion quickly and unambiguously, without the source having to wait for either a retransmit timer or three duplicate ACKs to infer a dropped packet. For bulk-data TCP connections, the delay for the retransmission of an individual packet is not generally an issue. For bulk-data TCP connections in wide-area environments, the congestion window is generally sufficiently large that the dropped packet is detected fairly promptly by the Fast Retransmit procedure. Nevertheless, for those cases where a dropped packet is not detected by the Fast Retransmit procedure, the use of ECN mechanisms can improve a bulk-data connection's response to congestion. If the source is delayed in detecting a dropped packet, perhaps due to a small congestion control window and a coarse-grained TCP timer, the source can lie idle. This delay, when combined with the global synchronization, can result in substantial link idle time (Floyd, 1994).

As the use of wireless networks grows, packet loses at the physical layer can be seen frequently. Packet losses are not always congestion notification. This false alarm causes the sender reduce its rate unnecessarily. In ECN, sender reduces its rate only if gets binary feedback about congestion from the receiver. Otherwise, it keeps

increasing the rate. Especially in wireless networks, ECN will maximize the link utilization.

4.5 DECbit

DECbit congestion avoidance scheme is the earliest example of congestion detection at gateways which is described by (Ramakrishnan & Jain, 1990). In DECbit scheme, each router monitors the queue size and explicitly notifies the sources when congestion is about to occur. This notification is implemented by setting a bit (DECbit) in the header of the packet that flows through the router. The router sets this bit if the average queue length is greater than or equal to one at the time the packet arrives at the router. When this explicit notification arrives to the sender in the header of the packet acknowledgement, the sources adjust its sending rate in order to avoid congestion. The sender uses the window based flow control mechanism. The sender updates their windows of data packets once every two round trip times. If at least half of the packets in the last window have the congestion indication bit set, then the window size is decreased exponentially, otherwise it is increased linearly. In other words, the sender decreases the congestion window by 0.875 times if 50 percent or more of the last windows worth of packets have the DECbit sent, otherwise the sender increments the congestion window by one packet. The queue length at the router as a function of time can be shown in Figure 4.2. The average queue length is calculated by the ratio of the area under the curve and the averaging interval. The use of DECbit mechanisms for the notification of congestion to the end nodes prevents unnecessary packet drops. This increases the performance and the utilization of the network. The main disadvantages of DECbit are averaging queue size for short periods of time and no difference between congestion detection and indication.

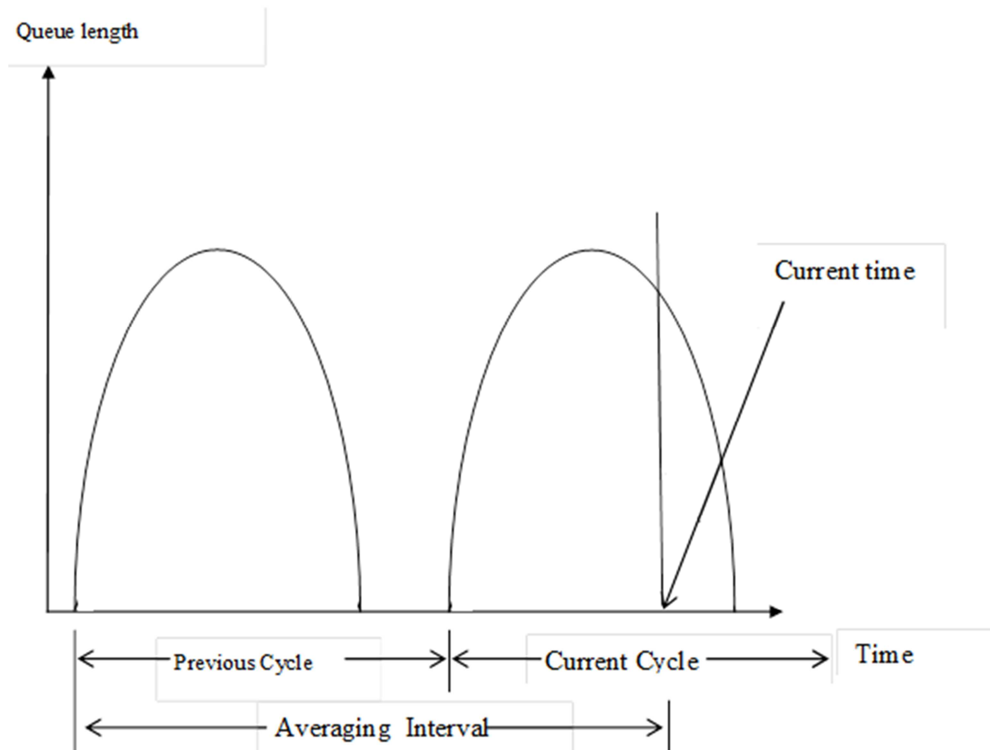

Figure 4.2 Queue length over time in DECbit.

4.6 Drop Tail & Drop Front on Full Algorithms

The drop tail algorithm is the simplest and most deployed algorithm, which is implemented by means of a First In First Out queue management. It simply drops the arriving packet if the buffer is full. Besides tail drop, an alternative queue disciplines drop front on full. Under the “drop front on full” algorithm, the router drops the packet at the front of the queue when the queue is full and a new packet arrives.

The biggest advantage of the drop tail algorithm is the easiness and the simplicity of the implementation, suitability to heterogeneity and its decentralized nature. However, it suffers several disadvantages such as, the higher delays suffered by packets when they go through longer queues. Both of these solve the lockout problem, but neither solves the full-queues problems. The algorithms do not perform well when the buffer is either long or short. If the buffer is long, a packet may

experience long delay. If the buffer is short, it is difficult for the algorithm to accommodate bursty traffic. These algorithms may cause global synchronization, which leads to the loss of throughput. If the queue is full or almost full, an arriving burst will cause multiple packets to be dropped. This can result in a global synchronization of flows throttling back, followed by a sustained period of lowered link utilization, reducing overall throughput. Both these two issues may lead the network to collapse.

Generally, drop tail algorithm is used as a baseline for comparing the performance of all the newly proposed IP level congestion control algorithms.

4.7 Random Drop Algorithm

The basic idea behind random drop algorithm is simple. For each arriving packet, if the buffer is full, the algorithm will randomly choose a packet from the queue to drop. This seems an improvement of the drop tail algorithm. However, it does not solve any of the disadvantages of the drop tail algorithm.

4.8 Early Random Drop Algorithm

Both the drop tail and random drop algorithms react to the congestion situation after it has already happened. Moreover, they both face some serious problems. Early random drop algorithm drops packets before the router's queues have been completely full. The early random drop algorithm is the first one falls into this category. The early random drop algorithm drops each packet arriving at the gateway with a fixed drop probability, if the queue length exceeds a certain drop level (threshold). This algorithm makes improvements over the drop tail and random drop algorithms but still with similar problems.

4.9 Random Early Detection Algorithm (RED)

To eliminate the Drop Tail disadvantages and to anticipate the source answers to incipient congestion situations, Floyd and Jacobson propose a mechanism called Random Early Detection (RED) (Floyd & Jacobson, 1993). RED is a popular example of active queue management (AQM) mechanisms (Braden *et al.,* 1998) (Feng *et al.,* 1999). RED is an active policy of queue management, which is now widely deployed and makes a decision to drop a packet randomly when the queue average length ranges between a minimum and a maximum threshold. The probability of packet dropping/marking is obtained from the average queue length accordingly to a linear law.

The basic idea of RED algorithm is to keep the average queue size low (and hence end-to-end delay) while allowing occasional bursts of packets in the queue. In the RED algorithm, the packet dropping probability is proportional to that connection's share of the throughput through the router. RED performs better than the drop tail algorithm because it has higher throughput and lower delays. It avoids global synchronization and has the ability to accommodate short bursts. It is easy to implement. It controls the average queue size even in the absence of non-adaptive sources. Because of its various advantages, in 1998, RED has been recommended as the standard of congestion avoidance mechanism in gateways. Pseudo code of RED algorithm can be found below.

```
For any arrival of packets,
        Calculate the average queue size (avg)
        If minTh ≤ avg ≤ maxTh
                Calculate packet dropping probability (Pa)
                Drop the arrived packet with probability Pa
        Else if maxTh ≤ avg
                Drop the arriving packet
```

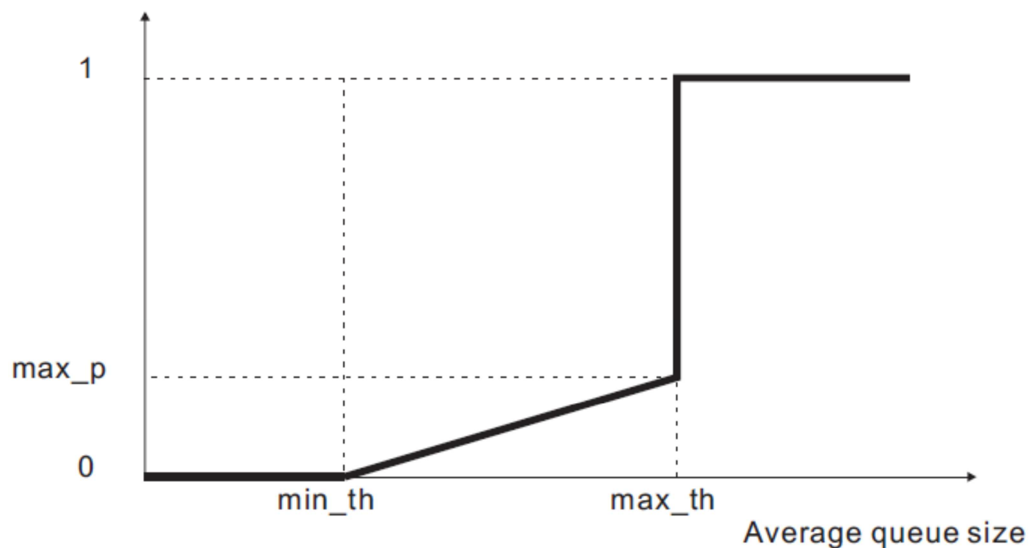

Figure 4.3 RED Algorithm.

The RED algorithm (see Figure 4.3) calculates the average queue size by assigning different weights (the exponential weight factor, a user-configurable value) to old value and current measure. This means the adoption of a low pass filter to reduce the high frequency variation of the instantaneous queue. For high values of n, the previous average becomes more important. A large factor smooths out occasional bursts and keeps the queue length low. The average queue size is unlikely to change very quickly. The RED algorithm will be slow to start dropping packets, but it may continue dropping packets for a time after the actual queue size has fallen below the minimum threshold. The slow moving average will accommodate temporary bursts in traffic. If the value of n gets too high, RED will not react to congestion. Packets will not be dropped by the RED algorithm. This would mean higher queuing delays.

On the other hand, if the maximum threshold is set to a low value, the average queue size is easily affected from the current queue size. The resulting average may fluctuate with changes in the traffic levels. In this case, the RED process responds quickly to long queues. Once the queue falls below the minimum threshold, the process will stop dropping packets. If the value of n gets too low, RED will overreact to temporary traffic bursts and drop traffic unnecessarily. This would mean a bad usage of the link because of severe buffer oscillations. From these considerations

follows that it is very difficult to find out the right trade-off, and it is hard to tune RED to achieve both high link utilization and low delay and packet losses.

Although RED is a big success in internet congestion control, it still suffers from some problems. Dropping packets from flows in proportion to their bandwidth does not always lead to fair bandwidth sharing. For example, if two TCP connections unevenly share one link, dropping one packet periodically from the low speed flow will almost certainly prevent it from claiming its fair share, even if the faster flow experiences more packet drops. RED is designed to work with adaptive flows. Non-adaptive flows can take over the link's bandwidth. A non-adaptive connection can force RED to drop packets at a high rate from all connections. RED heavily penalizes TCP flows and awards non-TCP flows.

In the last years, the active queue management policies have been object of a large interest in networking and several proposals (Feng *et al.,* 1999; Ott *et al.,* 1999; Clark & Fang, 1998) have been presented to find more effective control policies than RED. REM and PI (Hollot *et al.,* 2001) are proposed to solve the problems, which RED faces. Their solution is very similar to each other. REM aims to achieve a high utilization of link capacity, scalability, negligible loss and delay. As an improvement to RED, REM algorithm differentiates between the congestion measure of each router and the dropping probability. REM algorithm maintains a so-called variable price, which eliminates the dependence of the dropping probability from the current value of the queue size. The REM algorithm uses the current queue size and the difference from a desired value to calculate the dropping probability accordingly to an exponential law. A source calculates the price of the whole path using the knowledge of the total number of packets dropped on the path. The main disadvantages of REM algorithm is that it gives no incentive to cooperative sources and a properly calculated and fixed value of price variable must be known globally.

However, Lin &Morris (1997) define fragile TCP flows as those emanating from sources with either large round-trip delays or small send window sizes and robust TCP flows as having either short round-trip delays or large send windows. This description emphasizes a flow ability to react to indications of both increased and

decreased congestion at the bottleneck router. Their research indicates that RED is not fair when the router traffic includes both robust and fragile flows.

Floyd's original ECN paper (Floyd, 1994) shows the advantages of ECN over Red using both LAN and WAN scenarios with a small number of flows. Christiansen et al., (2000) use a LAN use a LAN test bed to emulate a large number of web clients accessing a web server through a RED router. They show that RED is difficult to tune for throughput and delay, and they conclude that minTh is the most significant RED parameter for performance tuning. Although they do consider flows with a large variation in round-trip time (RTT), they do not consider fairness in their analysis.

Bagal et al., (1999) compare the behavior of RED, ECN and a TCP rate-based control mechanism using traffic scenarios that include 10 heterogeneous flows. They conclude that RED and ECN provide unfair treatment when faced with either variances due to the RTTs of the heterogeneous flows or variances in flow drop probabilities (Kinicki & Zheng, 2001).

Kinicki & Zheng (2001) investigate the behavior and performance of RED with ECN congestion control mechanisms with many heterogeneous TCP Reno flows using the network simulation tool, NS2. By comparing the simulated performance of RED and ECN routers, they find that ECN does provide better goodput and fairness than RED for heterogeneous flows. However, when the demand is held constant, the number of flows generating the demand has a negative effect on performance. Meanwhile the simulations with many flows demonstrate that the bottleneck router's marking probability must be aggressively increased to provide good ECN performance. This investigation builds on these recent results to experiment with adaptive variations of ECN.

Kinicki & Zheng (2001) conduct simulation experiments on four Adaptive ECN (AECN) mechanisms. The results show these approaches can be used within an AQM framework to improve goodput and fairness for ECN routers. We are going to survey in detail most widely used congestion control algorithms on IP level in the next sections.

4.10 Weighted Random Early Detection (WRED)

In addition to the basic functionalities of RED, Weighted Random Early Detection (WRED) is used as the Cisco implementation of RED for standard Cisco IOS platforms (Cisco IOS Quality of Service Solutions Configuration Guide, Release 12.2). It provides the IP Precedence feature to provide quality of service, which gives the ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow. In WRED, drop decisions are made depending on IP precedence of the flow of interest. It drops the packets of the flows with lower priority when the queues start to be congested. Different classes of services can be configured with difference drop probabilities. WRED can be configured to ignore IP precedence when making drop decisions. This working mode is similar to classical RED algorithm.

WRED chooses packets from other flows to drop rather than the flows with IP precedence. Ordinary traffic flows with lower precedence have a higher drop rate, and therefore is more likely to be throttled back.

Global synchronization for responsive flows happens during period of congestion because packets start to be dropped all at once. Each source reduces their transmission rate at the same time when packet loss occurs. After that, sources increase their transmission rate when the congestion is cleared. WRED avoids the global synchronization as the congestion avoidance mechanism on the routers.

WRED is a congestion avoidance technique by randomly dropping the packets prior to congestion. Responsive flows respond to packet loses by decreasing their transmission rate until the congestion is cleared as RED performs. In addition to RED functions, WRED drops packets selectively based on IP precedence. Packets with a higher IP precedence are less likely to be dropped than packets with a lower precedence.

WRED reduces the chances of tail drop by selectively dropping packets when the output interface begins to show signs of congestion. By dropping some packets early rather than waiting until the queue is full, WRED avoids dropping large

numbers of packets at once and minimizes the chances of global synchronization. Thus, WRED allows the transmission line to be used fully at all times. In addition, WRED statistically drops more packets from large users than small. Therefore, traffic sources that generate the most traffic are more likely to be slowed down than traffic sources that generate little traffic.

WRED is only useful when the bulk of the traffic is TCP/IP traffic. With TCP, dropped packets indicate congestion, so the packet source will reduce its transmission rate. With other protocols, packet sources may not respond or may resend dropped packets at the same rate. Thus, dropping packets does not decrease congestion. WRED treats non-IP traffic as precedence 0, the lowest precedence. Therefore, non-IP traffic, in general, is more likely to be dropped than IP traffic. Figure 4.4 illustrates how WRED works.

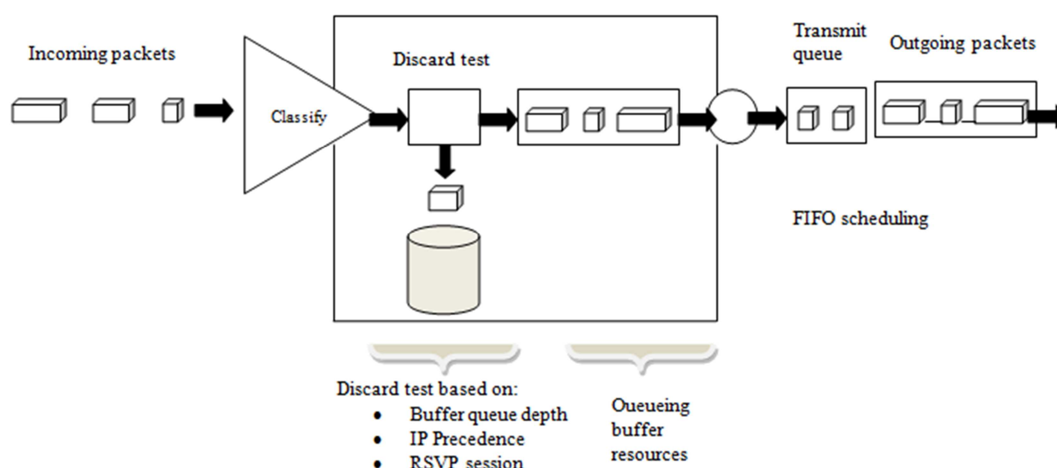

Figure 4.4 WRED algorithm.

WRED makes early detection of congestion possible and provides for multiple classes of traffic. It also protects against global synchronization. For these reasons, WRED is useful on any output interface where you expect congestion to occur. However, WRED is usually used in the core routers of a network, rather than at the edge of the network. Edge routers assign IP precedence to packets as they enter the network. WRED uses this precedence to determine how to treat

different types of traffic (Cisco IOS Quality of Service Solutions Configuration Guide, Release 12.2).

4.11 Distributed Weighted Random Early Detection (DWRED)

Distributed WRED is an implementation of WRED. DWRED provides VIP processing as well as all functionalities of WRED. In DWRED algorithm, when a new packet is queued at the router, the average queue is calculated. If the average queue size is less than the minimum threshold parameter, then the arriving packet is enqueued. If the average queue size is between the minimum threshold and the maximum threshold parameter, the packet is dropped with a packet drop probability. If the average queue size is greater than the maximum threshold, the packet is dropped.

The basic improvement in DWRED is to keep separate thresholds parameters on the queue size and weights for different IP precedence. This property provides different qualities of service for different traffic. Standard traffic may be dropped more frequently than premium traffic during periods of congestion.

4.12 Flow-Based Weighted Random Early Detection (Flow-Based WRED)

Flow-based WRED is an extension of WRED that provides fairness to all flows. Flow-based WRED classifies incoming traffic into flows according to the destination, source addresses and ports, and keeps state information about the flows, which have packets at the output queues. Flow-based WRED prevents each flow to occupy more than its permitted share of the resources using the state information and the classification of the flows. Flow-based WRED penalizes more the flows, which take over the resources available in the router. In order to provide fairness to all flows, flow-based WRED keeps a count of the number of the active flows, which have packets at the output queues. Using the number of active flows and the output

queue size, flow-based WRED calculates the number of buffers available per flow. Flow-based WRED permits each active flow to have a determined number of packets at the output queue by scaling the number of buffers available per flow by a configured factor. This scaling factor is common to all flows. Each flow is limited to have a maximum number of packets by the scaled number of buffers. The probability of a packet drop from a certain flow that exceeds the number of packets allowed by its per-flow limit increases.

4.13 Flow Random Early Drop Algorithm (FRED)

Flow Random Early Drop (FRED) is a modified version of RED algorithm. FRED algorithm aims to provide more resources for adaptive flows and to reduce the resource utilization of non-adaptive flows. The algorithm keeps state information of all flows currently present in the gateway. FRED keeps a parameter called strike for each flow. Strike is defined as the number of times that the flow has failed to respond to congestion notification. FRED penalizes flows with high strike values. For each flow, FRED keeps its queue length (qleni), the maximum queue length (maxq) and the minimum queue length (minq). It keeps the average queue size (avgcq) and calculates this parameter every time a new packet has been enqueued or serviced. At each packet arrival, FRED determines the flow of this packet. If the queue length of this flow exceeds the maximum queue length (maxq) for this queue, or this queue has a strike value bigger than 1 and its queue length is not less than the average queue length of all the queues (avgcq), this coming packet will be dropped. Both these two situations indicate this queue has tried more than once to break the maxq threshold or is trying to break the threshold, which makes it tend to be a misbehaving flow, so this coming packet will be dropped. Other than these two situations, FRED will act just like RED when the total queue length is less than minth (accept) or bigger than maxth (drop). When the total queue length falls between minth and maxth, it will check if the queue length of this current flow is larger than the average queue length of all the queues (avgcq) or the minimum queue length of this queue (minq). If so, FRED will perform random drop. If not, it will accept the coming packet. This

situation is based on the idea that flows with fewer packets queued in the buffer should be rewarded. When the algorithm sees a flow's queue length has not reached the average of all the flows or the minimum allowable queue length, FRED will reward this flow by accepting its packet. Figure 4.5 shows a comparison between the RED and FRED algorithm.

FRED algorithm provides fairness to all flows as the biggest advantage over RED. FRED awards the flows with less packets queued in the buffer than the average. FRED penalizes the flows with more packets. However, FRED keeps state information and some parameters for each distinct type of flows. FRED requires more memory to store the state information and processor capabilities to make computations about dropping decisions. FRED does not scale well. It also suffers the problem of setting the proper parameter values.

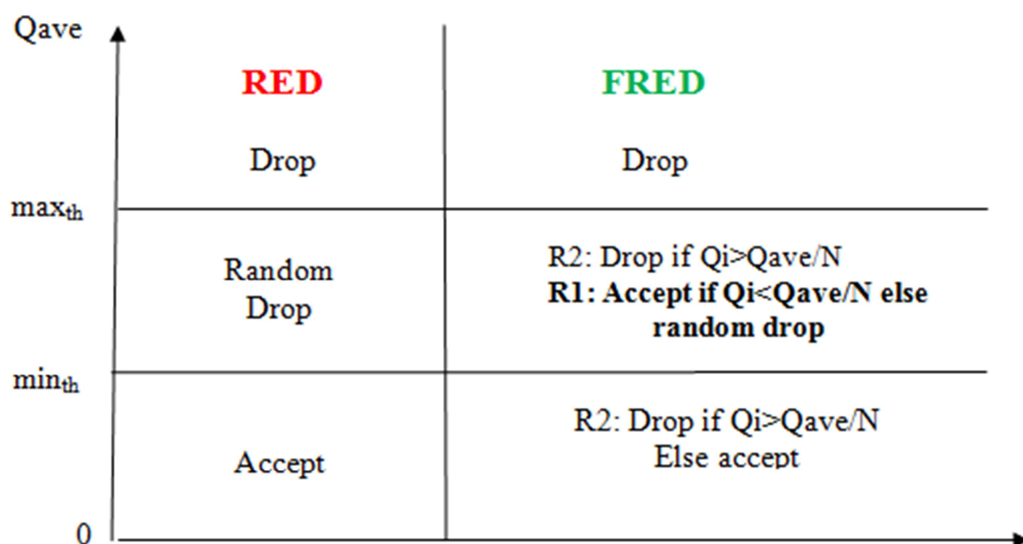

Figure 4.5 Comparison between the RED and FRED algorithm.

4.14 Stabilized RED Algorithm (SRED)

The stabilized RED algorithm (SRED) (Ott *et al.,* 1999) is a buffer management algorithm in order to make a stable buffer usage without affecting the number of flows in the router's buffer. SRED presents to improve performance and fairness of

the algorithms derived from RED idea. Unlike the FRED, SRED do not keep or analyze the state information of individual flows.

The main idea behind SRED is to estimate the number of active connections or flows in the buffer to adapt dropping probability accordingly. This estimation is based on a zombie list (Figure 4.6) where flows with high bandwidth are likely to be in the list. SRED creates an empty list and initializes the hit parameter to zero. At each packet arrival if the zombie list is not full, the packet's flow identifier (source address, destination address, etc.) is added to the list. At each packet arrival if the zombie list is full, the arriving packet is compared with a zombie in the list. If the arriving packet's flow matches the zombie, we declare a "hit". In that case, the Count of the zombie is increased by one, and the timestamp is reset to the arrival time of the packet in the buffer. If the two are not of the same flow, we declare a "no hit". In that case, with probability the flow identifier of the packet is overwritten over the zombie chosen for comparison. The SRED algorithm can be found in Figure 4.7.

A hit indicates that the flow has a higher probability of occupying the buffer and thus it might be unresponsive, a hit with a high-count value increases the probability, and a hit with a high count and a high total occurrence increases the probability even further.

The performance of the SRED algorithm is independent of the number of active connections. It does not need to have a special parameter for average queue length. It does not need to have state information. This reduces memory needs in the router.

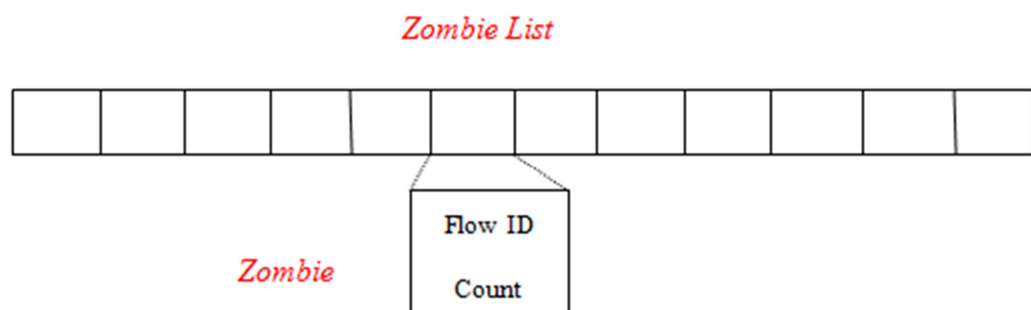

Figure 4.6 Zombie list.

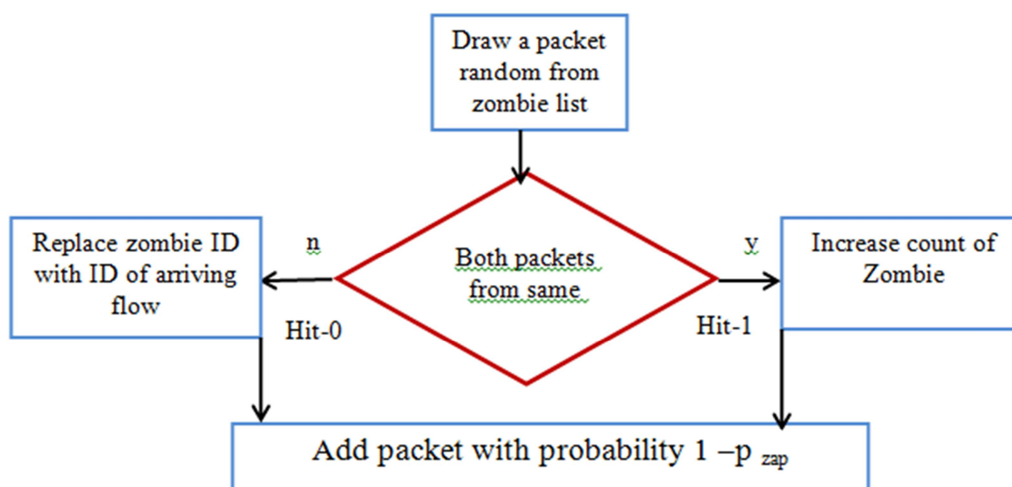

Figure 4.7 SRED algorithm.

However, SRED requires more computation especially for some high-speed links. Its control mechanism to reduce the resource usage of the non-adaptive flows is still not satisfactory enough. Its estimation of the number of active flows is not accurate when files have random sizes, rather than infinite size. The proper settings of the parameters are still difficult.

4.15 Choose & Keep for Responsive Flows, Choose & Kill for Unresponsive Flows (CHOKe)

Choose & Keep for Responsive Flows, Choose & Kill for Unresponsive Flows (CHOKe) (Pan *et al.,* 2000) is another modified version of RED. In addition to keeping the advantages of RED algorithm, the CHOKe algorithm aims to identify and penalize unresponsive flows with an easy implementation. It tries to reduce the the resource consumption of the flows, which consume the most resources. In the CHOKe algorithm, whenever a new packet arrives at the router, it updates the average queue size and compares the new value of the average queue size with the minimum threshold value. If the average queue size is less than the minimum threshold value, then the algorithm enqueues the incoming packet. If the average

queue size is greater than the minimum threshold, then it draws a packet randomly from the FIFO buffer and compares with the arriving packet. If both packets belong to the same flow, then both packets are dropped, else the incoming packet is admitted into the buffer with a probability that depends on the level of congestion. This probability is computed exactly the same as in RED. The CHOKe algorithm is shown in Figure 4.8.

CHOKe is a simple, easy to implement RED variant and stateless algorithm, which does not require any special data structure. It keeps the advantages of RED algorithm including the ability to avoid global synchronization, keeping the average buffer sizes low (low delays) and lack of bias against busty traffic. It improves the performance than RED by penalizing the unresponsive flows.

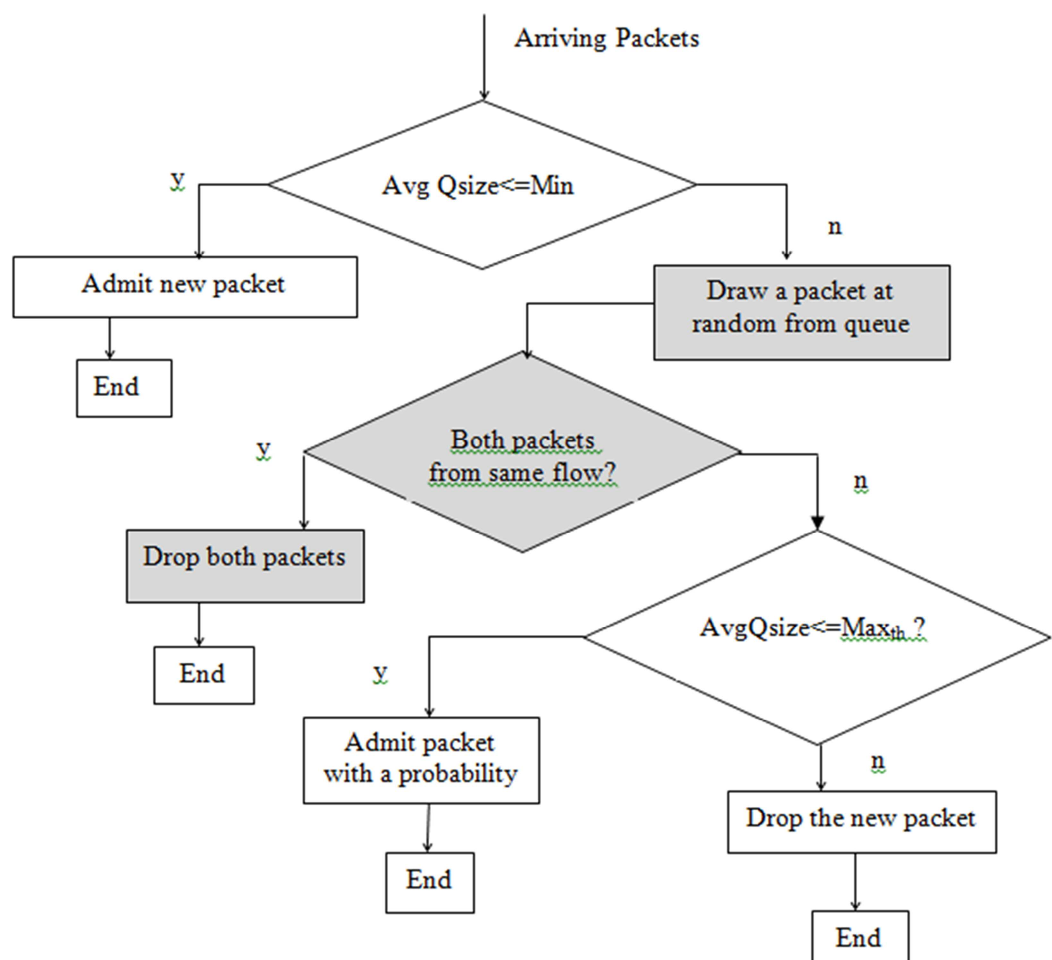

Figure 4.8 CHOKe algorithm.

However, this algorithm is not likely to perform well when the number of flows is large compared to the buffer space. The CHOKe algorithm still awards unresponsive flows like UDP. As the number of the flows in the router, the computations of the CHOKe get more expensive.

4.16 Comparison and Classifications of Major IP Level Algorithms

In this section, a comparison between RED and its variants including FRED, SRED, CHOKe is presented. In RED, there is no per flow treatment. RED does not keep or reserve buffers for flow information. Hence, RED is said to be the most unfair algorithm among its variants. RED is designed to work with adaptive flows. It rewards the non-adaptive flows like UDP. FRED algorithm is a complete per-flow treatment algorithm and fair for both adaptive and non-adaptive flows. It keeps flow information for each flow. It has better protection for adaptive flows and isolates non-adaptive greedy traffic. Both the SRED and CHOKe algorithms have per flow treatment for high-bandwidth flows (which tend to be non-adaptive flows or misbehaving flows). However, the SRED algorithm identifies those non-adaptive flows and does not take any effective actions to penalize those flows. Only the CHOKe algorithm not only identifies, but also penalizes the non-adaptive flows. SRED does not compute the average queue size giving less computational overhead.

RED and SRED are unfair whereas FRED and CHOKe are fairer. However, RED and CHOKe, FRED is too expensive, because it keeps state information for all the flows. The comparison between RED algorithm and its variants is summarized in Table 4.1.

Table 4.1 Comparision between RED and its variants

Algorithm	Fairness	Ease of Configuration	Buffer Size Requirement	Per-flow Information	Computational Overhead
RED	Bad	Bad	Bad	No	Bad
SRED	Good	Good	Good	Yes	Bad
FRED	Bad	Good	Good	No	Good
CHOKe	Bad	Good	Good	No	Good
DECBit	Good	Good	Good	No	Good

4.17 Summary on Active Queue Management Mechanisms

Additive Increase, Multiplicative Decrease principle is the basis of TCP's congestion control mechanism. IP level congestion control algorithms use this property of responsive flows to congestion notifications like TCP. Although the dominant protocol in internet is TCP, the number of non-adaptive flows like UDP in the internet increases. The congestion control algorithms should be able to identify and penalize the misbehaving flows to achieve a proportional fairness for adaptive flows.

Active queue management mechanisms use a drop probability depends on average amount of traffic, not on the specific short-term traffic statistics. The main advantage of active queue management mechanisms is to reduce the delay without sacrificing link utilization by absorbing bursts due to bursty sources or converging flows. With active queue management, it is easy to decide how many packets are accepted in a buffer for scheduling those accepted packets differently, depending on the nature of the applications. The main drawback of active queue management mechanisms is that it is not so easy to detect misbehaving flows that are not TCP friendly.

However, internet routers should implement active queue management mechanisms, which are in IP level to reduce average delay, to manage average queue length, to reduce packet dropping, and to avoid global synchronization in the internet. It is necessary to find effective mechanisms to deal with flows that are unresponsive to congestion notification or are responsive but more aggressive than TCP.

As we have investigated in this chapter, current active queue management mechanisms have their own advantages as well as they have their own drawbacks.

CHAPTER FIVE
DESCRIPTION OF OUR APPROACH

5.1 Orange, Our Proposed Algorithm

Multiple server queuing systems have wide applicability to the analysis of computer and communication systems. Although it simplifies the analysis of multiple server system, the homogeneity constraint is frequently violated in operational systems. The heterogeneity (different service rates) of the servers occurs in a communication network supporting that the communication channels (servers) can be affected by different transmission rates, processor speeds, available memories, etc. In this work, we address a special case of multiple server queuing systems where only two heterogeneous servers are involved. Under these circumstances, a queuing discipline designed for a system with two heterogeneous servers and a queue using a threshold type policy, referred to as Orange is defined and analyzed.

> The main motivation for using a threshold-based approach is that many systems incur significant server setup, usage, and removal costs. More specifically, under light loads, it is not desirable to operate unnecessarily many servers, due to incurred setup, and usage costs; on the other hand, it is also not desirable for a system to exhibit very long delays, which can result due to lack servers under heavy loads. One approach improving the cost performance ratio of a system is to react to changes in workload through the use of thresholds. For instance, one can maintain the expected job response time in a system at an acceptable level, and at the same time maintain an acceptable cost for operating that system, by dynamically adding or removing servers, depending on the system load. (Golubchik & Lui, 2002).

In this work, we consider to simulate a two heterogeneous servers and one queue with a threshold-based queuing system in order to achieve both higher throughput and lower queuing delays. We also consider finding a close relationship between the parameters of our algorithm using the mathematical analysis. The contributions of

this work are as follows. To the best of our knowledge, none of the works described earlier consider to use a virtual drop server to drop the incoming packets when the actual queue size (or average) exceeds a threshold level. The only adjustable parameter based on the changing conditions of the network is the service time of the virtual drop server. Since for many applications, this service time is not usable, we consider it an important and distinguishing characteristic of our work. We first aim to give an exact solution for computing the steady state probabilities of our model using Kolmogorov-Chapman equations. However, we feel that the exact solution is quite complicated and hard to be derived and not that necessary in practical applications. Moreover, thus, the main contribution of this work is an efficient solution of a threshold based queuing system with two heterogeneous servers and one queue.

By using the threshold type policy and the use of virtual drop server, we have proposed a new approach to drop or mark packets when the congestion will likely occur. We have intended to use an IP level congestion control proposal, which we call Orange. Orange will replace RED as an active queue management algorithm to decide which packets are to be marked to indicate a congestion condition. The idea behind Orange is similar to RED which also uses "early dropping" concept to regulate the flows before congestion occurs. Here, "early" refers the fact that actually as long as there is space in the queue buffer to place the incoming packet; we still chose to drop them to warn TCP friendly sources (responsive or adaptive) against that possible congestion situation.

In a threshold queuing discipline, customers (in our case, packets) are preferably routed to the faster server. Customers are allowed to queue up while the slower server remains idle until the queue size reaches a certain "threshold" value, at which a point a customer is removed from the queue and sent to the slower server for service. The threshold value becomes critical control parameter affecting system's overall performance, and facilitating optimal system control. The primary performance parameter is the mean number of customers in the system, and accordingly the average waiting time per packet. Optimization of the two heterogeneous servers problem is considered over an infinite time horizon with an average cost criterion. Although linear holding and service costs are considered, it is

generally assumed that there is no additional cost incurred to turn on or to turn off a server.

Orange is based on the idea of dropping packets, randomly whenever some conditions are met, that is equivalent of using an alternate server to the default link of that outgoing interface. Orange waits for a random amount of time after a dropping occurs before another one may be considered. This is the time equivalent of a service time sample of the "drop server".

Orange proposal's main idea relies on a single queue, two server M/M/2 model analyses. In this, first server is the link transmission element, and the second one is the unpreferred alternative link. The second one is used only when queue size exceeds a threshold. The optimum threshold value for such a system is analyzed by M. Kemal Şiş Ph.D. thesis (Şiş, 1994) and studied also in Gökhan Çatalkaya's Msc. Thesis (Çatalkaya, 2003). We expect to utilize this theoretical analysis in Orange's analytical evaluation giving the best operation point for responsive sources. Orange takes the mentioned optimum threshold policy to decide when the incoming packets will be "dropped through the virtual drop server".

Orange allows the incoming packet go to the queue for transmission if the queue size is below the threshold (Orange Limit). It drops the incoming packet and sets the timer if the timer is idle and the queue size is in between the Orange limit and the queue limit (maximum queue size). While the timer continues to be busy, Orange does not drop any incoming packet. Orange drops all incoming packets if the queue is full. One can refer to the Figure 5.1 for pseudo code of Orange algorithm.

```
If (queue limit) then
          Drop(packet)
Else
          If (Orange limit) then                    If (timer is idle) then
                                        Drop(packet)
                                        Begintimer()
                    Else
                                        Enque(packet)
                    End if
          Else
                    Enque(packet)
          End if
End if
```

Figure 5.1 Pseudo code of Orange algorithm.

The queuing model behind the Orange algorithm is mainly based on the M/M/2 queues. We intend to make a detailed analysis of the various types of M/M/2 queues. The M/M/2 case shown in Figure 5.2 is the simplest non-trivial case of a local model for a node in a network. In this type of network, for the traffic at the concerned node, there is only one final destination, but there are two different links by which the traffic can be carried toward the destination node. There may be several incoming links to the node; however, since all the traffic is destined to the same destination node it can all be stored in one queue. Arrivals to this queue are modeled as a Poisson arrival process (mean rate λ). Time spent on a link is modeled as exponentially distributed so links can be thought of as servers with exponentially distributed service times (mean rate μ). Therefore, the birth rate is always equal to λ, whereas the death rate depends on the state.

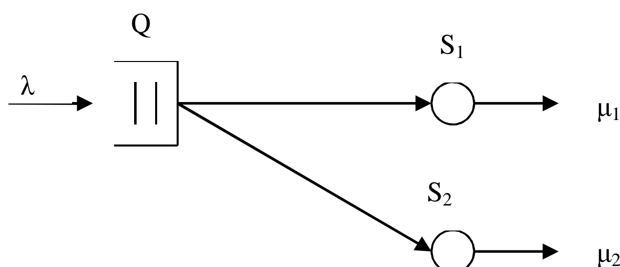

Figure 5.2 M/M/2 queue model.

5.2 Generic M/M/2 Queue Analysis

For the simplicity of the analysis for a generic M/M/2 case, we choose "$\mu_1=\mu_2=\mu$". We also study the detailed analysis of this case with heterogeneous servers where "$\mu_1 > \mu_2$". The intensity of the arrival process is "$\rho = \lambda/\mu$". The system is described by a Markov chain of the number of messages, as shown in Figure 5.2. Under the stability assumption, the Markov chain can be solved by means of the cut equilibrium conditions. The Markov chain modeling of this queue is shown in Figure 5.3.

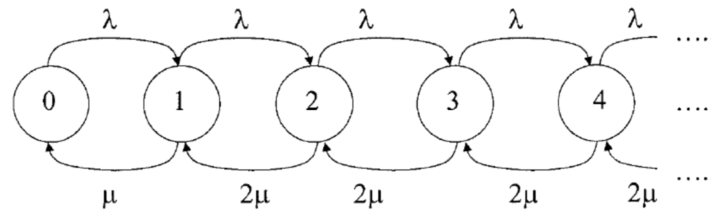

Figure 5.3: Continuous time Markov chain model for M/M/2 queue.

Cut 1 balance: $\lambda P_0 = \mu P_1 \Rightarrow P_1 = \rho P_0$

Cut 2 balance: $\lambda P_1 = 2\mu P_2 \Rightarrow P_2 = \frac{\rho^2}{2} P_0$

Cut 3 balance: $\lambda P_2 = 2\mu P_3 \Rightarrow P_3 = \frac{\rho^3}{2^2} P_0$

…

Cut n balance: $P_n = 2\left(\frac{\rho}{2}\right)^n P_0, n > 0$

We can finally write the normalization condition in order to obtain P_0:

$$P_0 = \frac{1}{1+\sum_{n=1}^{\infty}\frac{P_n}{P_0}} = \frac{1}{1+2\left(\sum_{n=0}^{\infty}\left(\frac{\rho}{2}\right)^n - 1\right)} = \frac{2-\rho}{2+\rho} \qquad \text{Eqn 5.1}$$

Note that "$P_0 > 0$" (i.e., the system is stable, because sometimes it can be idle at regime) entail "$\rho < 2$ Erlangs". The mean number of messages in the system can be obtained by means of the first derivative of the PGF of the state probability distribution, P (z). This PGF is obtained as a sum of different contributions of the type "$z_n P_n$"; hence, for the first derivative the value of the term "$z_0 P_0$" is not relevant. This is the reason why we use a P (z) related function, named P*(z), obtained as:

$$P^*(z) = \sum_{n=0}^{\infty} \frac{2-\rho}{2+\rho} z^n 2\left(\frac{\rho}{2}\right)^n = 2\frac{2-\rho}{2+\rho}\frac{1}{1-z\frac{\rho}{2}} = 4\frac{2-\rho}{2+\rho}\frac{1}{2-z\rho} \qquad \text{Eqn 5.2}$$

Note that P*(z) is not a PGF (P*(z=1) is not equal to 1), however, it can be used as a PGF in evaluating the first derivative and the mean number of messages in the queuing system, N:

$$N = \frac{dP*(z)}{dz}_{z=1} = 4\frac{2-\rho}{2+\rho}\frac{\rho}{(2-z\rho)^2}_{z=1} = \frac{4\rho}{4-\rho^2} \quad \text{Eqn 5.3}$$

The mean message delay to cross the queuing system (from the arrival to the transmission completion) can be obtained by means of the Little theorem as:

$$T = \frac{N}{\lambda} = \frac{4/\mu}{4-\rho^2} \quad \text{Eqn 5.4}$$

5.3 M/M/2 Queue Analysis with Heterogeneous Servers

We consider a variant of the M/M/2 queue where the service rates of the two servers are not identical. This would be the case, for example, in a heterogeneous multiprocessor system. The queuing structure is shown in Figure 5.4. Assume without loss of generality that "$\mu_1 \geq \mu_2$". Jobs wait in line in the order of their arrival. When both servers are idle, the faster server is scheduled for service before the slower one. The state diagram of the system is given in Figure 5.4.

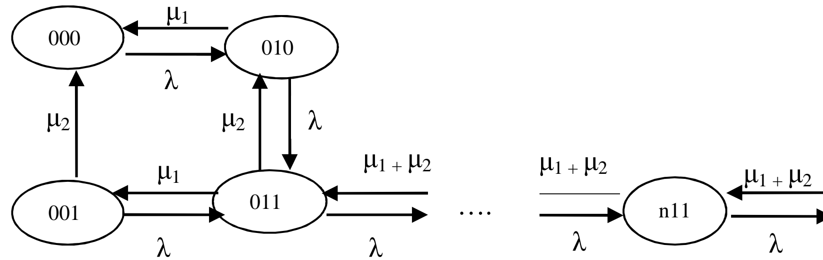

Figure 5.4: The State diagram for the M/M/2 heterogeneous queue.

Balance equations, in the steady state, can be written by equating the rate of flow into a state to the rate of flow out of that state:

$$\lambda P_{(000)} = \mu_1 P_{(0)} + \mu_2 Q_{(001)}$$
$$(\lambda+\mu_1)P_{(0)} = \mu_2 Q_{(0)} + \lambda P_{(000)} \quad \text{Eqn 5.5}$$
$$(\lambda+\mu_2)Q_{(001)} = \mu_1 Q_{(0)}$$

$$(\lambda+\mu_1+\mu_2)Q_{(0)} = (\mu_1+\mu_2)Q_{(1)} + \lambda Q_{(001)} + \lambda P_{(0)}$$

$$(\lambda+\mu_1+\mu_2)Q_{(n)} = (\mu_1+\mu_2)Q_{(n+1)} + \lambda Q_{(n-1)} \quad n > 1$$

The traffic intensity for this system is

$$\rho = \frac{\lambda}{\mu_1+\mu_2} \qquad \text{Eqn 5.6}$$

The previous form equation is similar to the balance equation of a birth-death process.

$$P_{(n)} = \frac{\lambda}{\mu_1+\mu_2} P_{(n-1)} \qquad n > 1 \qquad \text{Eqn 5.7}$$

By repeated use of this equation, we have

$$P_{(n)} = \rho P_{(n-1)} = \rho^{n-1} P_{(1)} \qquad n > 1 \qquad \text{Eqn 5.8}$$

From the above equations, we can obtain by solving linear equations by Gaussian elimination using elementary row operations:

$$\begin{bmatrix} -\mu_2 & 0 & (\lambda+\mu_1) \\ -\mu_1 & (\lambda+\mu_2) & 0 \\ 0 & \mu_2 & \mu_1 \end{bmatrix} \begin{bmatrix} Q_{(0)} \\ Q_{(001)} \\ P_{(0)} \end{bmatrix} = \begin{bmatrix} \lambda P_{(000)} \\ 0 \\ \lambda P_{(000)} \end{bmatrix}$$

The augmented matrix form;

$$\begin{bmatrix} -\mu_2 & 0 & (\lambda+\mu_1) & \lambda P_{(000)} \\ -\mu_1 & (\lambda+\mu_2) & 0 & 0 \\ 0 & \mu_2 & \mu_1 & \lambda P_{(000)} \end{bmatrix}$$

$$\begin{bmatrix} -\mu_2 & 0 & (\lambda+\mu_1) & \lambda P_{(000)} \\ 0 & (\lambda+\mu_2) & -\frac{\mu_1}{\mu_2}(\lambda+\mu_1) & -\frac{\mu_1}{\mu_2}\lambda P_{(000)} \\ 0 & \mu_2 & \mu_1 & \lambda P_{(000)} \end{bmatrix}$$

$$\begin{bmatrix} -\mu_2 & 0 & (\lambda+\mu_1) & \lambda P_{(000)} \\ 0 & (\lambda+\mu_2) & -\frac{\mu_1}{\mu_2}(\lambda+\mu_1) & -\frac{\mu_1}{\mu_2}\lambda P_{(000)} \\ 0 & 0 & \frac{\mu_1(\lambda+\mu_1)}{(\lambda+\mu_2)}+\mu_1 & (\frac{\mu_1}{(\lambda+\mu_2)}+1)\lambda P_{(000)} \end{bmatrix}$$

$$\left(\frac{\mu_1(\lambda+\mu_1)}{(\lambda+\mu_2)}+\mu_1\right) P_{(0)} = \left(\frac{\mu_1}{(\lambda+\mu_2)}+1\right)\lambda P_{(000)} \qquad \text{Eqn 5.9}$$

$$\left(\mu_1(2\lambda+\mu_1+\mu_2)\right) P_{(0)} = \left(\lambda+\mu_2+\mu_1\right)\lambda P_{(000)} \qquad \text{Eqn 5.10}$$

$$P_{(0)} = P_{(000)}\frac{\lambda}{\mu_1}\frac{(\lambda+\mu_1+\mu_2)}{(2\lambda+\mu_1+\mu_2)} = P_{(000)}\frac{\lambda}{\mu_1}\frac{\left(1+\frac{\lambda}{\mu_1+\mu_2}\right)}{(1+\frac{2\lambda}{\mu_1+\mu_2})} \qquad \text{Eqn 5.11}$$

$$P_{(0)} = \frac{1+\rho}{1+2\rho}\frac{\lambda}{\mu_1}P_{(000)} \qquad \text{Eqn 5.12}$$

By substitution, we can have the probability equation of the other initial states

$$Q_{(0)} = \frac{\rho}{1+2\rho}\frac{\lambda(\lambda+\mu_2)}{\mu_1\mu_2}P_{(000)} \qquad \text{Eqn 5.13}$$

$$Q_{(001)} = \frac{\rho}{1+2\rho}\frac{\lambda}{\mu_2}P_{(000)} \qquad \text{Eqn 5.14}$$

Now, observing that

$$\left[\sum_{n\geq 0} Q_{(n)}\right] + Q_{(001)} + P_{(0)} + P_{(000)} = 1 \qquad \text{Eqn 5.15}$$

We have

$$(\sum_{n\geq 0}\rho^n)Q_{(0)} + P_{(000)}\left[\frac{\rho}{1+2\rho}\frac{\lambda}{\mu_2}+\frac{1+\rho}{1+2\rho}\frac{\lambda}{\mu_1}+1\right] = 1 \qquad \text{Eqn 5.16}$$

Or

$$\frac{1}{1-\rho}\frac{\rho}{1+2\rho}\frac{\lambda(\lambda+\mu_2)}{\mu_1\mu_2}P_{(000)} + P_{(000)}\left[\frac{\rho}{1+2\rho}\frac{\lambda}{\mu_2}+\frac{1+\rho}{1+2\rho}\frac{\lambda}{\mu_1}+1\right] = 1 \qquad \text{Eqn 5.17}$$

From which we get

$$P_{(000)} = \left[1 + \frac{\lambda(\lambda+\mu_2)}{\mu_1\mu_2(1-\rho)(1+2\rho)}\right]^{-1} \qquad \text{Eqn 5.18}$$

The average number of jobs in the system may now be computed by observing that the number of customers in the system in state (K, s_1, s_2) is K + s_1 + s_2. Therefore, the number of average jobs is given by,

$$E[N] = \left(\sum_{k\geq 0} kP_{(k)} + Q_{(0,0,1)} + \sum_{k\geq 0}(k+1)Q_{(k)}\right) \qquad \text{Eqn 5.19}$$

$$E[N] = P_{(0)} + Q_{(0,0,1)} + \sum_{k\geq 0}(k+1)Q_{(k)} \qquad \text{Eqn 5.20}$$

$$E[N] = P_{(0)} + Q_{(0,0,1)} + \sum_{k\geq 1} Q_{(k)} + \sum_{k\geq 1} kQ_{(k)} \qquad \text{Eqn 5.21}$$

$$E[N] = 1 - P_{(0,0,0)} + Q_{(0)}\sum_{k=1}^{\infty} k\rho^{k-1} \qquad \text{Eqn 5.22}$$

$$E[N] = 1 - P_{(0,0,0)} + \frac{Q_{(0)}}{(1-\rho)^2} \qquad \text{Eqn 5.23}$$

$$E[N] = \frac{1}{A(1-\rho)^2} \qquad \text{Eqn 5.24}$$

Where;

$$A = \left[\frac{\mu_1\mu_2(1+2\rho)}{\lambda(\lambda+\mu_2)} + \frac{1}{1-\rho}\right] \qquad \text{Eqn 5.25}$$

5.4 M/M/2 Queue Analysis with a Threshold K=1

We consider a variant of the M/M/2 queue where the service rates of the two servers are different where "$\mu_1 \geq \mu_2$". There is a threshold on using the slower server which is "K=1". This means that the slower server will be busy if and only if there is more than one customer in the queue. The queuing structure is shown in Figure 5.5.

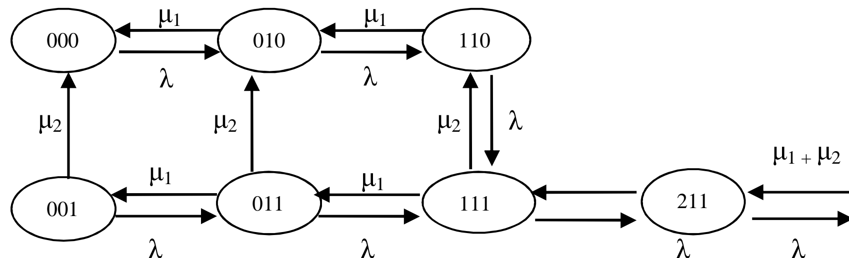

Figure 5.5 Continuous time Markov chain for M/M/2 queue with threshold K

$$\lambda P_{(000)} = \mu_1 P_{(0)} + \mu_2 Q_{(001)}$$
$$(\lambda+\mu_1)P_{(0)} = \mu_2 Q_{(0)} + \lambda P_{(000)} + \mu_1 P_{(1)}$$
$$(\lambda+\mu_2)Q_{(001)} = \mu_1 Q_{(0)} \qquad \text{Eqn 5.26}$$
$$(\lambda+\mu_1)P_{(1)} = \mu_2 Q_{(1)} + \lambda P_{(0)}$$
$$(\lambda+\mu_1+\mu_2)Q_{(0)} = \lambda Q_{(001)} + \mu_1 Q_{(1)}$$

Now, we have five unknowns and five equations, we can write these equations in matrix form.

$$\begin{bmatrix} \mu_1 & \mu_2 & 0 & 0 & 0 \\ (\lambda+\mu_1) & 0 & -\mu_2 & -\mu_1 & 0 \\ 0 & (\lambda+\mu_2) & -\mu_2 & 0 & 0 \\ -\lambda & 0 & 0 & (\lambda+\mu_1) & -\mu_2 \\ 0 & -\lambda & (\lambda+\mu_1+\mu_2) & 0 & -\mu_1 \end{bmatrix} \begin{bmatrix} P_{(0)} \\ Q_{(001)} \\ Q_{(0)} \\ P_{(1)} \\ Q_{(1)} \end{bmatrix} = \begin{bmatrix} \lambda P_{(000)} \\ \lambda P_{(000)} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

For K = 0, we have 3 initial states, for K = 1, we have 5 initial states, so for K, we will have 3 + (2*K) states.

$$(\lambda+\mu_1+\mu_2)Q_{(1)} = (\mu_1+\mu_2)Q_{(2)} + \lambda Q_{(0)} + \lambda P_{(1)} \qquad \text{Eqn 5.27}$$
$$(\lambda+\mu_1+\mu_2)Q_{(n)} = (\mu_1+\mu_2)Q_{(n+1)} + \lambda Q_{(n-1)} \quad n > 1$$

The sum of all probabilities equals unity,

$$\left[\sum_{n\geq 0} Q_{(n)}\right] + P_{(0)} + Q_{(001)} + Q_{(0)} + P_{(1)} + Q_{(1)} = 1 \qquad \text{Eqn 5.28}$$

One can find a symbolic solution for initial states by Gaussian elimination using elementary row operations. So all states probabilities can be written in terms of $P_{(000)}$. Then we can find the average number of customer in the system in terms of λ, μ_1, μ_2. By Little's Theorem, we can find the average delay per packet.

5.5 M/M/2 Queue Analysis with a Threshold K

Within the framework of this work, we consider an M/M/2 queuing model where a threshold on the queue size controls to use or not to use the slower server. In the simplest non-trivial case, we may consider a switching node with two out-going links and one type of traffic load stored in a single outbound buffer. This may seem too basic for any router application; however, applications that are more meaningful may also be decomposed into such several basic nodes.

The mathematical formulation of the performance criterion is a model for measuring the actual physical performance of the network. The dynamic local strategy that we consider will be based on a performance criterion to model the congestion reducing capability of the network at the node level. Furthermore, this performance criterion, we claim, is a good approach to minimizing a delay (source-to-destination) objective in the overall network. Lin and Kumar (Lin and Kumar, 1984) showed that the optimal policy, for an M/M/2 node model and for a cost function defined as the mean sojourn time of customers in the system, is of "threshold type". Continuous time Markov chain for a M/M/2 Queue with threshold K is given in Figure 5.6.

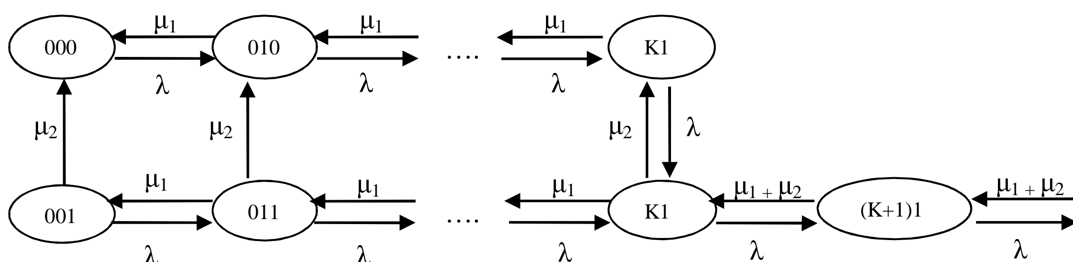

Figure 5.6 Continuous time Markov chain for M/M/2 queue with threshold K.

We assume here that "$\mu_1 \geq \mu_2$" and "$\mu_1 + \mu_2 > 1$".

We can write the state probability equations below;

Initial states;

$$P_{(000)}\lambda = P_{(0)}\mu_1 + Q_{(001)}\mu_2$$

$$Q_{(001)}(\lambda+\mu_2) = Q_{(0)}\mu_1$$

$$P_{(0)}(\lambda+\mu_1) = P_{(000)}\lambda + Q_{(0)}\mu_2 + P_{(1)}\mu_1 \qquad \text{Eqn 5.29}$$

$$Q_{(0)}(\lambda+\mu_1+\mu_2) = Q_{(001)}\lambda + Q_{(1)}\mu_1$$

Intermediate states;

$$P_{(1)}(\lambda+\mu_1) = P_{(0)}\lambda + Q_{(1)}\mu_2 + P_{(2)}\mu_1$$

$$Q_{(1)}(\lambda+\mu_1+\mu_2) = Q_{(0)}\lambda + Q_{(2)}\mu_1$$

$$P_{(2)}(\lambda+\mu_1) = P_{(1)}\lambda + Q_{(2)}\mu_2 + P_{(3)}\mu_1 \qquad \text{Eqn 5.30}$$

$$Q_{(2)}(\lambda+\mu_1+\mu_2) = Q_{(1)}\lambda + Q_{(3)}\mu_1$$

$$P_{(3)}(\lambda+\mu_1) = P_{(2)}\lambda + Q_{(3)}\mu_2 + P_{(4)}\mu_1$$

$$Q_{(3)}(\lambda+\mu_1+\mu_2) = Q_{(2)}\lambda + Q_{(4)}\mu_1$$

…….

$$P_{(n)}(\lambda+\mu_1) = P_{(n-1)}\lambda + P_{(n+1)}\mu_1 + Q_{(n)}\mu_2 \quad (0 < n < K) \qquad \text{Eqn 5.31}$$

$$Q_{(n)}(\lambda+\mu_1+\mu_2) = Q_{(n-1)}\lambda + Q_{(n+1)}\mu_1 \quad (0 < n < K)$$

$$P_{(K)}(\lambda+\mu_1) = P_{(K-1)}\lambda + Q_{(K)}\mu_2 \quad (n = K) \qquad \text{Eqn 5.32}$$

$$Q_{(K)}(\lambda+\mu_1+\mu_2) = Q_{(K-1)}\lambda + Q_{(K+1)}(\mu_1+\mu_2) + P_{(K)}\lambda \quad (n = K)$$

$$Q_{(n)}(\lambda+\mu_1+\mu_2) = Q_{(n-1)}\lambda + Q_{(n+1)}(\mu_1+\mu_2) \quad (K < n < \infty) \qquad \text{Eqn 5.33}$$

The balance equation of a birth-death process is

$$P_{(n)}(\lambda+\mu_1+\mu_2) = P_{(n-1)}\lambda + P_{(n+1)}(\mu_1+\mu_2) \quad n > 1 \qquad \text{Eqn 5.34}$$

By repeated use of this balance equation, we have

$$P_{(n)} = \frac{\lambda}{(\mu_1+\mu_2)} P_{(n-1)} \qquad \text{Eqn 5.35}$$

$$P_{(n)} = \frac{\lambda}{(\mu_1+\mu_2)} P_{(n-1)} = \left(\frac{\lambda}{(\mu_1+\mu_2)}\right)^{n-1} P_{(1)} \quad n > 1 \qquad \text{Eqn 5.36}$$

For the states K < n < ∞

$$\left[\sum_{n=K+1}^{\infty} Q_{(n)}\right] + A = 1 \qquad \text{Eqn 5.37}$$

$$\left[\sum_{n=K+1}^{\infty} \rho^{n}\right] Q_{(K+1)} + A = 1 \qquad \text{Eqn 5.38}$$

$$\rho^{n} = \frac{\lambda}{(\mu_1+\mu_2)} \qquad \text{Eqn 5.39}$$

$$\frac{1}{(1-\rho)} Q_{(K+1)} + A = 1 \qquad \text{Eqn 5.40}$$

For the solution of the initial states, namely A where (0 =< n < K); we have a square matrix with dimension of (3+2*K) like in the following representation.

$$\begin{bmatrix} \mu_2 & \mu_1 & 0 & 0 & 0 & \cdots & \cdots \\ (\lambda+\mu_2) & 0 & -\mu_1 & 0 & 0 & \cdots & \cdots \\ 0 & (\lambda+\mu_1) & -\mu_2 & -\mu_1 & 0 & \cdots & \cdots \\ -\lambda & 0 & (\lambda+\mu_1+\mu_2) & 0 & -\mu_1 & \cdots & \cdots \\ 0 & -\lambda & 0 & (\lambda+\mu_1) & -\mu_2 & -\mu_1 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \begin{bmatrix} Q_{(001)} \\ P_{(0)} \\ Q_{(0)} \\ P_{(1)} \\ Q_{(1)} \\ \cdots \\ P_{(K-1)} \\ Q_{(K-1)} \end{bmatrix} = \begin{bmatrix} \lambda P_{(000)} \\ 0 \\ \lambda P_{(000)} \\ 0 \\ 0 \\ 0 \\ 0 \\ \cdots \end{bmatrix}$$

We have calculated for the results for some small K (especially 0, 1) values relating the rational functions. These results could also be calculated by using a general-purpose symbolic-numerical-graphical mathematics software product namely Maple or Mathematica. However, one can never see a clear pattern, in this form of representation.

By using z-transform analysis, we can derive the exact expressions. However, the exact expressions for P, and Q, for even moderate K values, become very cumbersome. Because of this, the exact optimum value of K involves a very complicated implicit formula and it is not necessary to find the exact equation for optimum value of K. Instead, we can use an empirical solution for it. For this

purpose, we write these equations in Z-domain further to be solved. These equations can be represented by the following equations in the Z-domain.

Original equations;

$$P_{(n)}(\lambda+\mu_1)=P_{(n-1)}\lambda + P_{(n+1)}\mu_1 + Q_{(n)}\mu_2 \quad (0 < n < K) \quad \text{Eqn 5.41}$$
$$Q_{(n)}(\lambda+\mu_1+\mu_2) = Q_{(n-1)}\lambda+Q_{(n+1)}\mu_1 \quad (0 < n < K)$$

Equations in the Z-domain;

$$P(z)(\lambda+\mu_1)=\lambda z^{-1}P(z) + \mu_1\big(zP(z) - zP(0)\big) + \mu_2 Q(z) \quad (0 < n < K) \text{ Eqn 5.42}$$
$$Q(z)(\lambda+\mu_1+\mu_2) = \lambda z^{-1}Q(z)+\mu_1\big(zQ(z) - zQ(0)\big) \quad (0 < n < K)$$

Original equations;

$$Q_{(n)}(\lambda+\mu_1+\mu_2) = Q_{(n-1)}\lambda+Q_{(n+1)}(\mu_1 + \mu_2) \quad (K < n < \infty) \quad \text{Eqn 5.43}$$

Equations in the Z-domain;

$$Q(z)(\lambda+\mu_1+\mu_2) = \lambda z^{-1}Q(z)+(\mu_1 + \mu_2)\ \big(zQ(z) - zQ(0)\big) \quad (K < n < \infty)$$

Eqn 5.44

These equations can be used to find the solution by using the inverse z-transform. After finding the solution of the above equations, the probability of each state as a function of the initial state P(000) could be determined. This initial state could be a function of (λ, μ_1, μ_2,K). Using these probability equations, the average number of packets in the system and by Little's formula the average delay per packet in terms of μ_2,K could be determined. Exact expressions for P, and Q, for even moderate K values, become very cumbersome and are out of scope of this work. Instead, it is enough to use an empirical solution in order to find a relationship between system parameters. One of the main contributions of our work is to minimize the average delay per packet using the optimum value of threshold K.

5.6 Two Server Queue One Server Idle Below a Threshold

M/M/2 queue with a threshold is studied by (Şiş, 1994). In his work, First passage time to an idle period (FPTIP) is studied. FPTIP value is derived as a function of $\mu 1$, $\mu 2$, λ. Here, we want to evaluate a formula for the average queue size and waiting time for the same system. The exact solution becomes cumbersome and is not efficient and necessary for M/M/2 system with a threshold. Moreover, the main contribution of this work is to find a direct relationship between the threshold value and the service time pair that give the minimum waiting delays per packet instead of finding an exact solution. For this aim, we consider to use the equations, which Morrison (1990) has studied in his paper.

In his work, Morrison finds an efficient solution of a threshold based queuing system with two heterogeneous servers and one queue. In his work, for the sake of simplicity, he considers a birth-death queuing system with two exponential servers with mean rates "μ", and Poisson arrivals with mean rate is "$\lambda < 2\mu$", first in first out queuing discipline, unlimited buffer size of the bottleneck queue. Both servers are in use when the number of the customer in the system is more than a threshold level "c". Only one server is in use when the number of the customers in the system is less than "$c + 1$". Thus, the service rate of both servers is equal to each other; it is not important which server becomes idle. This system reduces to the generic M/M/2 case when "c" is one. So it is necessary to study the cases where "$c > 1$" for a non-trivial generalization.

The equilibrium probabilities of the number in the system are known (Kleinrock, 1975) and the mean waiting and sojourn times may be obtained from these by Little's formula. The system can be summarized as a single server system where the mean service rate is "μ" when there are less than "$c + 1$" customers in the system, and "2μ" when there are more than "c" customers are in the system.

From Morrison's study, we easily state that, the equilibrium probability P_0 that there is no customer in the system is

$$\frac{1}{P_0} = \begin{cases} \frac{1-(\lambda / \mu)^c}{1-(\lambda / \mu)} + \frac{2\mu}{(2\mu-\lambda)}\left(\frac{\lambda}{\mu}\right)^c & for\, \lambda \neq \mu \\ c+2 & for\, \lambda = \mu \end{cases}$$ Eqn 5.45

The equilibrium probability P_i that there are "i" customers in the system is

$$P_i = \begin{cases} P_0(\lambda / \mu)^i & for\, 0 \leq i \leq c \\ P_0(\lambda / \mu)^c(\lambda /2\mu)^{i-c} & for\, i \geq c \end{cases}$$ Eqn 5.46

The mean waiting and sojourn times are given by

$$\langle W \rangle = P_0\left(\frac{\lambda}{\mu}\right)^c \left\{\frac{\mu}{(\mu-\lambda)^2}[(\mu /\lambda)^{c-1} - 1] + (c-1)\left[\frac{1}{(2\mu-\lambda)} - \frac{1}{(\mu-\lambda)}\right] + \frac{\lambda}{(2\mu-\lambda)^2}\right\} \qquad for\, \lambda \neq \mu$$ Eqn 5.47

$$\langle W \rangle = \frac{P_0}{2\lambda}c(c+1) \qquad for\, \lambda = \mu$$ Eqn 5.48

$$\langle T \rangle = P_0\left(\frac{\lambda}{\mu}\right)^c \left\{\frac{\mu}{(\mu-\lambda)^2}[(\mu /\lambda)^{c} - 1] + (c)\left[\frac{1}{(2\mu-\lambda)} - \frac{1}{(\mu-\lambda)}\right] + \frac{2\mu}{(2\mu-\lambda)^2}\right\} \qquad for\, \lambda \neq \mu$$

Eqn 5.49

$$\langle T \rangle = \frac{P_0}{2\lambda}(c^2+3c+4) \qquad for\, \lambda = \mu$$ Eqn 5.50

We can adapt the Morrison's results into our case by substituting "$2\mu = \mu_1 + \mu_2$", "$\mu = \mu_1$", and "$c = K + 1$" in the above equations. Therefore, the mean waiting and sojourn times of M/M/2 with a threshold K case are found as

$$\langle W \rangle = P_0\left(\frac{\lambda}{\mu_1}\right)^{K+1} \left\{\frac{\mu_1}{(\mu_1-\lambda)^2}[(\mu_1 /\lambda)^{K} - 1] + (K)\left[\frac{1}{((\mu_1+\mu_2)-\lambda)} - \frac{1}{(\mu_1-\lambda)}\right] + \frac{\lambda}{((\mu_1+\mu_2)-\lambda)^2}\right\} \qquad for\, \lambda \neq \mu_1$$ Eqn 5.51

$$\langle W \rangle = \frac{P_0}{2\lambda}(K+1)(K+2) \qquad for\, \lambda = \mu_1$$ Eqn 5.52

$$\langle T\rangle = P_0\left(\frac{\lambda}{\mu_1}\right)^{K+1}\left\{\frac{\mu_1}{(\mu_1-\lambda)^2}\left[(\mu_1/\lambda)^{K+1}-1\right]+(K+1)\left[\frac{1}{((\mu_1+\mu_2)-\lambda)}-\frac{1}{(\mu_1-\lambda)}\right]+\frac{(\mu_1+\mu_2)}{((\mu_1+\mu_2)-\lambda)^2}\right\} \quad for\ \lambda \neq \mu_1 \qquad \text{Eqn 5.53}$$

$$\langle T\rangle = \frac{P_0}{2\lambda}\left((K+1)^2+3(K+1)+4\right) \quad for\ \lambda = \mu_1 \qquad \text{Eqn 5.54}$$

Where

$$\frac{1}{P_0} = \begin{cases} \frac{1-(\lambda/\mu_1)^{K+1}}{1-(\lambda/\mu_1)} + \frac{(\mu_1+\mu_2)}{((\mu_1+\mu_2)-\lambda)}\left(\frac{\lambda}{\mu_1}\right)^{K+1} & for\ \lambda \neq \mu_1 \\ K+3 & for\ \lambda = \mu_1 \end{cases} \qquad \text{Eqn 5.55}$$

Those derived formulas are used to justify the simulation results, which is available in next chapter. The equations are valid in a system where the Poisson arrivals and exponentially distributed service rates are applied. However, most of the flows in today's networking world consist of responsive flows like TCP, which adjust their sending rate according to the congestion indications from the network. Therefore, memory-less arrival process cannot be a realistic assumption. In systems, which have the memory-less property, the time distribution until the next event is the same regardless of how much time has passed since the last event, and the average time until the next event is the same as the average inter-event time. This property is also a direct consequence of the complete randomness of the Poisson process; what happens in the current interval is independent of what has happened in the previous interval.

The main goal of active queue management algorithms is to warn TCP friendly sources about the incoming congestion situation so that they will be able to reduce their sending rate to prevent the network to get in congestion collapse. The main objection of our proposed algorithm is to provide better conditions (high throughput and low per packet delays) for the networks where not only the constant bit rate sources but also the responsive sources are available. Therefore, it is meaningful to provide practically an empirical formula to determine the best operating point for Orange having its system parameters (threshold and service time) tuned for such conditions.

CHAPTER SIX
MATERIALS AND METHODS

6.1 Constructing the Simulation Environment

The experimental investigation aims to verify the results of our proposed study with the mathematical analysis. One important contribution of our work is to validate the proposed model experimentally. Network simulators have been extensively used to validate and evaluate the performance of network protocols.

6.1.1 Introducing NS (Network Simulator)

The NS (Network Simulator) is a good example of a widely used, public domain discrete event simulator targeted at network protocol research. It was originally implemented at LBL (Lawrence Berkeley Laboratory) and is currently being extended as part of the DARPA-funded VINT (Virtual Internet Test-bed) project at USC ISI.

NS is an event driven, packet level network simulator, developed by University of California Berkeley. Version 1 of NS was developed in 1995 and version 2 was released in 1996. Version 2 included a scripting language called Object oriented Tcl (OTcl). (Further, we mean the NS-2 by using NS.) It is an open source software package available for both Windows (via Cygwin) and Linux platforms. NS has many and expanding uses including:

- To evaluate the performance of existing network protocols.
- To evaluate new network protocols before use.
- To run large scale experiments not possible in real experiments.
- To simulate variety of IP networks.

NS is popularly used in the simulation of routing and multicast protocols. It implements network protocols such as TCP and UDP, traffic source behavior such as FTP, Telnet, Web, CBR and VBR, router queue management mechanism such as Drop Tail, RED and CBQ, routing algorithms such as Dijkstra, and more. NS also implements multicasting and some of the MAC layer protocols for LAN simulations. The NS project is now a part of the VINT Project that develops tools for simulation results display, analysis and converters that convert network topologies generated by well-known generators to NS formats.

NS implements the following features:

- Router queue management techniques DropTail, RED, CBQ,
- Multicasting
- Simulation of wireless networks
    - Developed by Sun Microsystems + UC Berkeley (Daedalus Project)
    - Terrestrial (cellular, adhoc, GPRS, WLAN, BLUETOOTH), satellite
    - IEEE 802.11 can be simulated, Mobile-IP, and adhoc protocols such as DSR, TORA, DSDV and AODV.
- Traffic source behavior- WEB, CBR, VBR
- Transport agents- UDP/TCP
- Routing
- Packet flow
- Network topology
- Applications- Telnet, FTP, Ping
- Tracing packets on all links/specific links

The simulation engine of NS is implemented in C++, and uses the object-oriented version of Tool Command Language developed at MIT (OTcl) as its front end. Accessing the NS library through this front end is possible with commands or scripts that are written in the Tcl language.

With an interpreter style of code execution, the NS program interprets each line of user script in a Tcl program during execution time and produces the output in the form of a formatted text file. Post processing is performed in order to filter particular data throughout the simulation time. Tools are required, such as “gnuplot” as to create graphs or Network Animator (nam) as to display an animated visualization of the NS simulations. (See Figure 6.1)

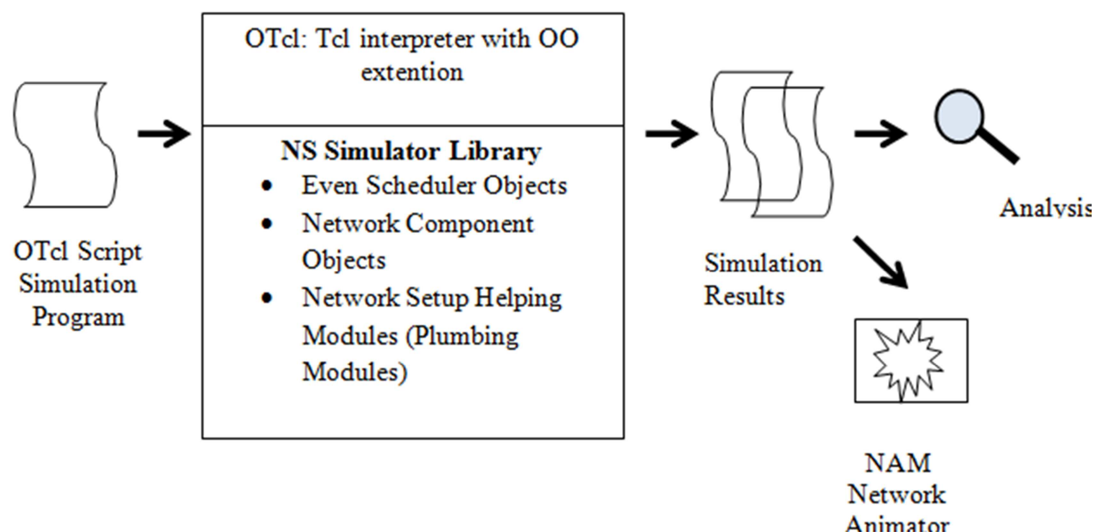

Figure 6.1 Simplified user's view of NS.

Besides the interpreted style of code execution, NS gives us a chance to write all commands in a script file for further execution. An OTcl script will do the following.

- Initiates an event scheduler.
- Sets up the network topology using the network objects.
- Tells traffic sources when to start/stop transmitting packets through the event scheduler.

Another major component of NS besides network objects is the event scheduler. An event in NS is a packet ID that is unique for a packet with scheduled time and the pointer to an object that handles the event. The event scheduler in NS performs the following tasks:

- Organizes the simulation timer.
- Fires events in the event queue.
- Invokes network components in the simulation.

Depending on the user's purpose for an OTcl simulation script, simulation results are stored as trace files, which can be loaded for analysis by an external application:

- A NAM trace file (filename.nam) for use with the Network Animator Tool
- A Trace file (filename.tr) for use with XGraph or TraceGraph.

NS is written in C++ with OTcl interpreter as a front end. For efficiency reason, NS separates the data path implementation from control path implementations. NS is a Tcl interpreter to run Tcl Scripts. By using C++/OTcl, the network simulator is completely object oriented.

- Scripting Language Tcl - Tool Command Language (pronounced "tickle")
- System Programming Language (C/C++)

In terms of lines of source code, NS was written with 100k lines of C++ code, 70k lines of Tcl code and 20k of documentation.

TclCL is the language used to provide a linkage between C++ and OTcl. Toolkit Command Language (Tcl/OTcl) scripts are written to set up/configure network topologies. TclCL provides linkage for class hierarchy, object instantiation, variable binding and command dispatching. OTcl is used for periodic or triggered events.

Event scheduler and basic network component objects are written and compiled with C++. These compiled objects are made available to the OTcl interpreter through an OTcl linkage that creates a matching OTcl object for each of the C++ objects and makes the control functions and the configurable variables specified by the C++ object act as member functions and member variables of the corresponding OTcl object. It is also possible to add member functions and variables to a C++ linked OTcl object. Architectural view of NS can be found in Figure 6.2.

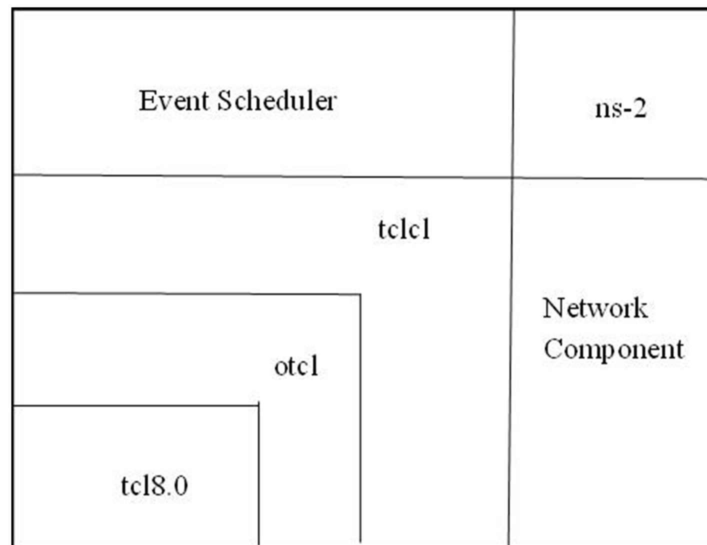

Figure 6.2 Architectural view of NS.

NS is designed to run from on most UNIX based operating systems. It is possible to run NS on Windows machines using Cygwin (A linux emulator for windows). Standard development packages like “make”, “gcc” and their dependencies must be available to compile the sources.

In NS, the network is constructed using nodes, which are connected using links. Events are scheduled to pass between nodes through the links. Nodes and links can have various properties associated with them. Agents can be associated with nodes and they are responsible for generating different packets (e.g. TCP agent or UDP agent). The traffic source is an application, which is associated with a particular agent (e.g. ping application). NS is very structured. This is illustrated in the Figure 6.3.

Links are required to complete the topology. In NS, the output queue of a node is implemented as part of the link, so when creating links the user also has to define the queue type. NS supports numerous queue types including FIFO, RED (Random Early Detection), Drop Tail, FQ (Fair Queuing), SFO(Stochastic).

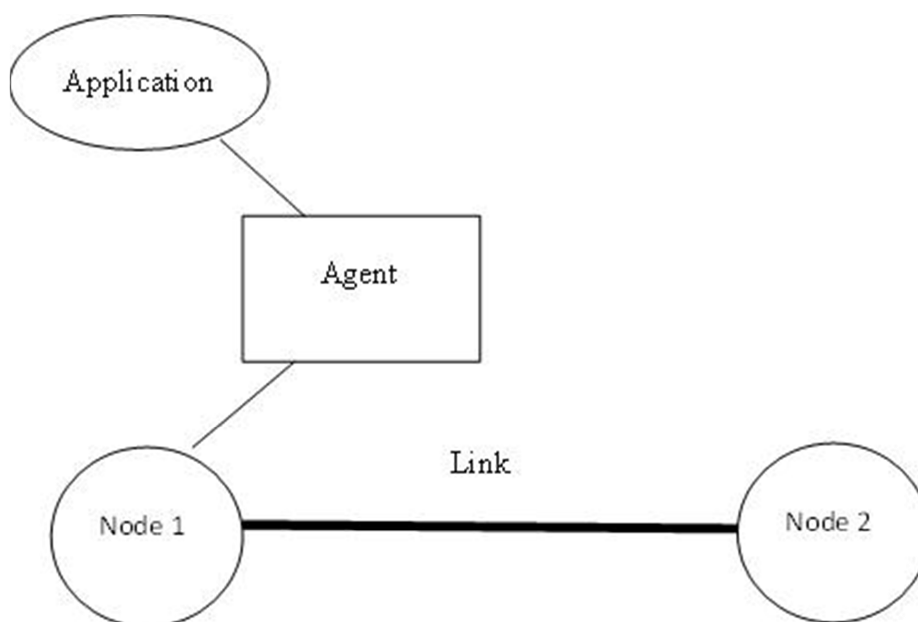

Figure 6.3 Two nodes, a link, an agent and an application.

Figure 6.4 shows the construction of a simplex link in NS. If a duplex link is created, two simplex links will be created, one for each direction. In the link, packet is first enqueued at the queue. After this, it is either dropped (passed to the Null Agent and freed there), or dequeued (passed to the Delay object which simulates the link delay). Finally, the TTL (time to live) value is calculated and updated.

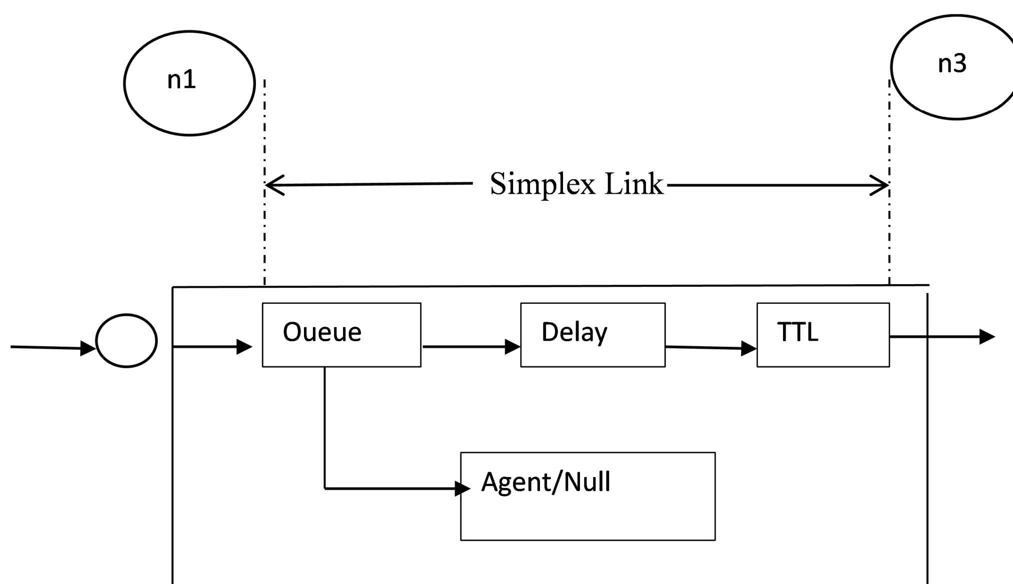

Figure 6.4 Link in NS.

Traffic generation in NS is based on the objects of two classes, the class Agent and the class application. Every node in the network that needs to send or receive traffic has to have an agent attached to it. On top of an agent runs an application. The application determines the kind of traffic that is simulated. There are two types of agents in NS: UDP and TCP agents

The Agent/LossMonitor can monitor number of packets transferred, as well as packets lost. A procedure can be scheduled to poll the LossMonitor every T seconds and obtain throughput information.

Four traffic applications are available in NS. They go on top of a UDP agent to simulate network traffic.

<u>CBR (Constant Bit Rate):</u> A CBR traffic object generates traffic according to a deterministic rate. Packets are of a constant size.

<u>Exponential:</u> Traffic is determined by an exponential distribution. Packets are a constant size. This produces an on/off distribution. Packets are sent at a fixed rate during on periods. No packets are sent during off periods.

<u>Pareto:</u> The distribution for traffic generation is taken from a pareto on/off distribution. This is generally used to generate aggregate traffic that exhibits long-range dependency.

<u>Traffic Trace:</u> Traffic is generated according to a trace file. The binary file must contain 2 * 32 fields in network (big-endian) byte order. The first field contains the time in milliseconds until next packet is generated. The second field contains the length in bytes of the next packet. The method filename of the Tracefile class associates a trace file with the Tracefile object.

In order to be able to calculate the results from the simulations, the data has to be collected. NS supports two primary monitoring capabilities: traces and monitors. The traces enable recording of packets whenever an event such as packet drop or arrival occurs in a queue or a link. The monitors provide a means for collecting quantities, such as number of packet drops or number of arrived packets in the queue. The

monitor can be used to collect these quantities for all packets or just for a specified flow (a flow monitor).

First, the output file is opened and a handle is attached to it. Then the events are recorded to the file specified by the handle. Finally, at the end of the simulation, the trace buffer has to be flushed and the file has to be closed. This is usually done with a separate finish procedure. If links are created after these commands, additional objects for tracing (EnqT, DeqT, DrpT and RecvT) will be inserted into them. These new objects will then write to a trace file whenever they receive a packet.

This trace file contains enqueue operations ('+'), dequeue operations ('-'), receive events ('r') and drop event ('d'). The fields in the trace file are: type of the event, simulation time when the event occurred, source and destination nodes, packet type (protocol, action or traffic source), packet size, flags, flow id, source and destination addresses, sequence number and packet id.

Tracing all events from a simulation to a specific file and then calculating the desired quantities from this file for instance by using Perl or Awk and Matlab is an easy and suitable way when the topology is relatively simple and the number of sources is limited. However, with complex topologies and many sources this way of collecting data can become too slow. The trace files will also consume a significant amount of disk space.

There are several advantages to using NS. First, it provides flexibility by allowing simulation of protocols at different layers of the network protocol stack. Because it has been widely used by the networking research community, it has accumulated considerable "common knowledge" in the form of contributed modules implementing different network protocols. Providing a comparable simulation framework is indeed one of VINT's goals. For example, at the routing layer, it supports both unicast and multicast. At the transport layer, NS includes implementations of different versions of TCP. NS can also be used in emulation mode; this allows the simulator to interface to a live network by accepting/injecting traffic from/into a real network. NS's emulation facility is especially useful since it

serves as an intermediate step between pure simulation and full-blown live experimentation.

Trace driven simulations have been widely used in systems validation and performance evaluation in different areas of computer science and electrical engineering. They have been particularly important in computer networking research. Seminal work in protocol design and performance evaluation has made extensive use of trace driven simulations. More recently, performance evaluation and tuning of World Wide Web protocols and applications have also employed trace driven simulation techniques.

As a result, public domain packet traces have been made available to the Internet research community. The National Laboratory for Applied Network Research (NLANR) maintains a collection of packet traces (containing only packet header information) that are publicly accessible. The Internet Traffic Archive is another well-known source of publicly available Internet packet traces.

We implement the proposed algorithm within NS and drive our simulations using several packet traces that are representative of traffic on the Internet. Packet traces serve as input to the proposed traffic models and intrusion detection and response control systems. Simulations subject the proposed models and control systems to various traffic patterns and boundary conditions. Simulation results provide feedback into the modeling tasks and become keys to understanding and tuning the proposed model.

6.1.2 Post Simulation Analysis

With an interpreter style of code execution, the NS program interprets each line of user script in a Tcl program during execution time and produces the output in the form of a formatted text file. Post processing is performed in order to filter particular data throughout the simulation time. Because of this, some text processing tools like Awk, Perl are required to produce the statistics information from NS's trace files.

We prefer to use Awk, which is one of the most interesting text processing languages used for NS trace analysis. Awk text-processing programming language is a useful and simple tool for manipulating text for tallying information from text files, creating reports from the results, and performing mathematical operations on files of numeric data. Awk text-processing language is useful for such tasks as:

- Tallying information from text files and creating reports from the results.
- Adding additional functions to text editors like “vi”.
- Translating files from one format to another.
- Creating small databases.
- Performing mathematical operations on files of numeric data.

Awk is not really well suited for extremely large, complicated tasks. It is also an “interpreted” language that is, an Awk program cannot run on its own, it must be executed by the Awk utility itself. That means that it is relatively slow, though it is efficient as interpretive languages go, and that the program can only be used on systems that have Awk.

To analyze trace files generated by the TCL simulation scripts, we develop some Awk scripts to calculate average throughput, average delay and average jitter for a given flow of the topology and to produce instantaneous throughput information, which can then be used to plot graphs (e.g. using Gnuplot) (see the Appendices D, E and F) .

6.1.3 Integrating ORANGE to NS

One important contribution of our work is to validate the proposed model experimentally because of the heuristic involved. We choose NS as our simulation platform and install latest version of all in one package of NS (ns-allinone-2.30) to Linux Ubuntu Version 10.4 (32 bit). Our goal is to implement the proposed study within NS and evaluate their performance analysis. Simulation results provided

feedback into the modeling tasks and become keys to understanding and tuning the proposed model.

Some simulation experiments have been performed for our proposed Orange algorithm in order to keep throughput high but average packet delay (thus, average queue sizes) low compared to DropTail and RED algorithms. We have developed, C++ codes for Orange algorithm in the NS core, Tcl codes for creating the sample topology, and the Awk scripts to post process the output trace files, which can be found in the Appendices. In order to integrate a new protocol into NS, we need to write a new class derived from the original RED code, which is a part of the NS queue library. However, the main difference between RED and Orange take place when a new packet comes to the queue. We have also added a new function, which is a decision mechanism when to drop the incoming packet.

A new C++ orange class has been derived from NS's default class "queue". Following lines show its definition.

```
class Orange : public Queue {
        public:
                Orange();
                ~Orange()     {
                                        delete q_;}
        protected:
                void reset();
                int command(int argc, const char*const* argv);
                void enque(Packet*);
                Packet* deque();
                PacketQueue *q_; /*Underlying FIFO queue*/
                int drop_front_;
                int summarystats;
                void print_summarystats();
                int qib_;
                int mean_pktsize_;
                int orange_limit;
                int queue_limit;
                double orange_timer;
                double ExpOrangeTimer;
                double lastDrop;
                int byOrange;
                int byDroptail;
};
```

Constructor of this class initializes a new queue and variables, which are defined on protected zone by default values in "ns-2.30/tcl.lib/ns-default.tcl". Method timeout() sets a new timer with timestamp which equals to orange_timer. Method t_status() returns the status of the timer, possible return values are TIMER_IDLE, TIMER_PENDING and TIMER_HANDLING. Method command parses tcl command which is passed from the oTcl object. Methods deque() and enque(), deque and enque the queue by the methods of the class "Queue".

There are a few requirements in order to integrate Orange into NS. After we develop orange.cc and orange.h files (see Appendices A, B, C), we must copy them into the folder "ns/queue", "edit ns/tcl.lib/ns-default.tcl", and add following lines.

```
Queue/Orange set drop_front_ false
Queue/Orange set summarystats_ false
Queue/Orange set queue_in_bytes_ false
Queue/Orange set mean_pktsize_ 500
Queue/Orange set queue_limit_ 30
Queue/Orange set orange_limit_ 20
Queue/Orange set orange_timer_ 50
```

Edit makefile, add "queue/orange.o" somewhere where the queues are. That is enough, the only thing we must do is to recompile the NS by using "make clean", "make depend", and "make" commands.

After these modifications, we develop some Awk scripts in order to analyze the experiments output. Some information resulted from these scripts are, simulation start time, simulation end time, number of sent packets, number received packets, number of dropped packets, average throughput, average delay, etc. In addition to this, we develop an extra Awk script that counts the drops according to the drop reason for which whether the drop is caused by physical limits of the queue buffer or the drop is caused earlier by the algorithm itself. We have ability to get this information by modifying the implementations of both Red and Orange algorithm in NS core. Finally, we develop a script (see Appendix D), which helps us to draw size of the bottleneck queue over time, and calculates the average queue size from the output file of the NS simulations. All scripts we develop help us to observe clearly the results of the experiments.

6.2 Topology Alpha with Poisson Sources

After making necessary modifications in NS, we decide to verify and test the simulation environment and our proposed algorithm with the theoretically computed values. For this aim, we decide to construct and use the topology in Figure 6.5, namely Topology Alpha. We have written the Tcl code that generates the topology, and the necessary Poisson Sources where its packet sizes at the input of the ongoing link is to be set to a finite value before the simulation starts. Source code of the script can be found in the Appendix G. In this set of simulation, especially to compare the simulation results with the mathematical calculation, we need to use Poisson sources, which are not available in the NS's default installation. In order to use the Poisson sources, we patched the default NS installation to include the Poisson Sources with the source codes by Kostas Pentikousis (2004).

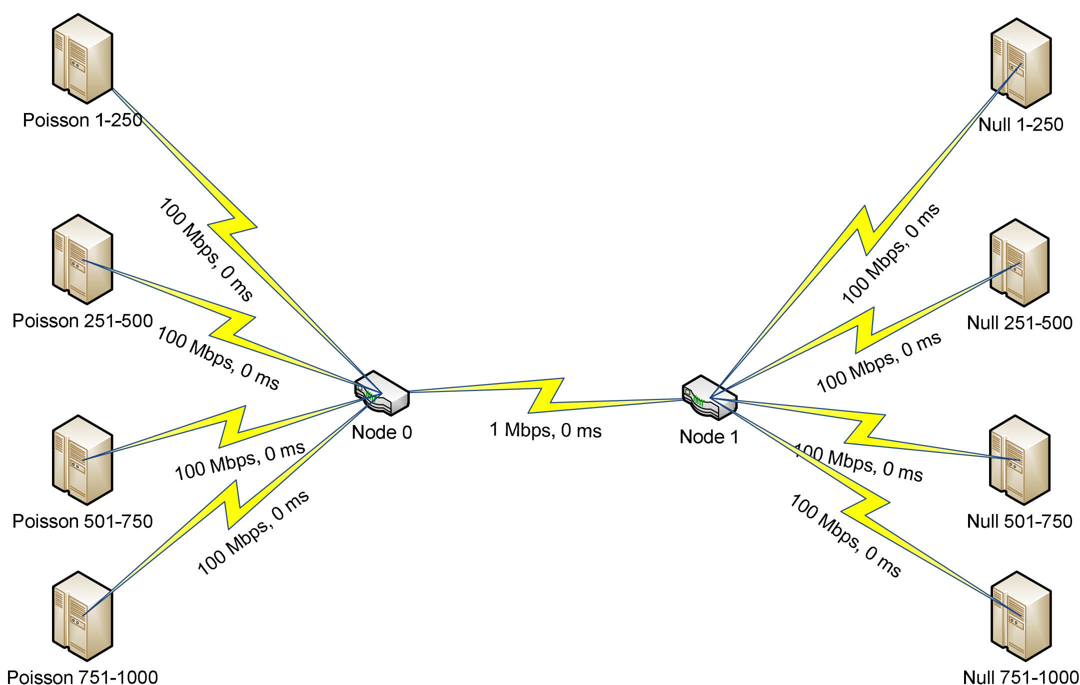

Figure 6.5 Topology Alpha.

However, Poisson sources in patched NS generate packets of constant size that we have to set in the beginning of the simulation. In order to overcome this lack of NS, we make a large number of Poisson sources involve in the simulation by setting the packet size of each traffic source is different and determined by random number

generator that generates exponentially distributed packet sizes with an average value. Aggregating Poisson sources in this way generates a traffic source with exponentially distributed sending rates, and packet sizes with a mean value, which is assumed to be in the mathematical analysis.

Note that, we have four different nodes in simulation topology. Because, in NS, the total number of agents we can connect to a node is 256, so we have to connect the 1000 sources and destinations to 4 different nodes where each node has 250 different sources.

6.2.1 Simulation of a M/M/1/K Queue

We start our work from a well-known simple model in a finite capacity system, namely, the “M/M/1/K” queue. “M/M/1/K” queue is the most popular finite capacity system where the customers arrive according to a Poisson process at rate “λ” and receive exponentially distributed service with a mean service rate “μ” from a single server. The difference from the “M/M/1” queue system is that at most “K” customers are allowed to be queued into the system. In case the queue is full of packets, the incoming packet is simply discarded. In communication systems, discarded packets are called “lost” packets and the system is called “loss” system. There is a special case when “K = 1” where the capacity of the queue is only one packet. It means that only one packet is allowed to be queued if the server is idle. The “block calls rejected” is used and the system is referred to as a queue with truncation like in telephone systems. (See Figure 6.6)

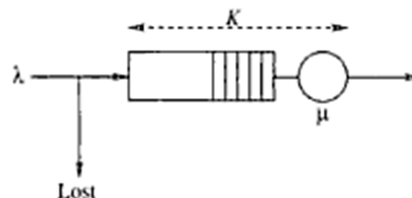

Figure 6.6 M/M/1/K queue system.

The packet loss rate of an "M/M/1/K" queue is given by the following equation (Gupta *et al.,* 2009).

$$p = \frac{(1-\rho)\rho^{K+1}}{1-\rho^{K+1}} \qquad \text{Eqn 6.1}$$

where

$$\rho = \frac{\lambda}{\mu} \qquad \text{Eqn 6.2}$$

The average queue length is given by the following equation (Gupta *et al.,* 2009).

$$q = \frac{\rho}{1-\rho} - \frac{(K+1)\rho^{K+1}}{1-\rho^{K+1}} \qquad \text{Eqn 6.3}$$

In order to test our modifications in NS, we would like to simulate "M/M/1/K" queue and compare the results with the above formulas. This queue requires that the service time, and the size of the packets to be exponentially distributed. Aggregating Poisson sources in this way generates a queue close to "M/M/1/K" queue.

In this set of simulation, we want to aggregate traffic to generate a total arrival rate "λ = 1200 packets/s", and average packet size equal to 100 bytes. Suppose that we aggregate 1000 Poisson sources. The packet arrival rate of each source is then 1.2 pps. The packet size of each source is set by sampling an exponential random variable of average 100 bytes. Each link capacity between the source and the bottleneck node should be high enough to prevent unnecessary packet drops. To be at the safe side, 100 Mbps link capacity with zero link delays is good enough to prevent unnecessary packet drops. Please note that since the sources generate packets regardless of the capacity of the link in front of them, link capacities do not have any effect on the simulation as long as they are high enough. Poisson sources with different packet sizes compete for a bottleneck link with zero delay and the capacity of 1Mbps. Thus, the average service rate of the link can be computed for 100 bytes packets as 1250 pps. The buffer size "K" is 100 packets. Average queue size is then computed using the above formula for the simulation parameters as 22.33 packets.

We make a simulation of ten seconds using the Topology Alpha with the above parameters and obtain some results after using post text processing tools. We use the DropTail queue at the bottleneck queue in the simulation, and find the average queue size from the simulation is 22.65 packets, which corresponds to the computed value. This result shows that our simulation environment is constructed and tested properly.

Table 6.1 Simulation results of M/M/1/K queue

Queue Type	Sent Packets	Arrival to Router	Arrival to Destination	Average Delay ms	Average Throughput Kbps	Average Queue Size
Drop Tail	13.139	12.341	12204	52.91	951.55	22.65

Simulation results can be found in Table 6.1. In Table 6.1, “Sent Packets” gives the number of the packets that have already injected to the network from the traffic source to their destinations. “Arrival to Router” gives the number of the packets that have already received by the bottleneck queue. The difference between “Sent Packets” and “Arrival to Router” comes from the fact that when the time simulation stops, there have been still some packets in transit. “Arrival to Destination” gives the number of the packets that have already delivered to their final destination. In other words, it gives the number of the packets that have been successfully transmitted. The difference between “Arrival to Router” and “Arrival to Destination” comes from the fact that some of the packets could be dropped along their way to their final destination due to the early drop of the queuing algorithm applied or the force drop of the queuing algorithm when its internal buffer gets full. “Average Delay” gives the average delay per packet in milliseconds. “Average Throughput” gives the average value of the throughput of the bottleneck link during simulation. Note that, this value includes the number of the retransmitted packets (if any). It means that, it is not “Goodput” which is defined as the number of packets that are exchanged between the applications that use the network excluding the retransmitted packets. “Average Queue Size” gives the average queue size in packets.

6.2.2 Effect of Orange on Simulation's Performance

In order to test our algorithms performance, we make a simulation of Topology Alpha for another ten seconds and obtain some results after using post text processing tools. After integrating a new queue type into NS, namely Orange, we use this object when creating links in Tcl layer of the NS similar to the following statement:

"$ns duplex-link $node0 $node1 1Mb 20ms Orange".

As an example, this command creates a duplex link between two nodes, which have 1 Mbps bandwidth, 20 ms delay, and Orange queue type. The physical buffer size of the bottleneck queue is fixed at 120 packets based on published rules of thumb for accommodating the network bandwidth delay product. Queue size is calculated in packets not in bytes.

In order to test our algorithm's performance, we have simulated the Topology Alpha for different queue types (DropTail, Orange, and RED). When Orange is applied to the above topology, the queue model can be considered as an "M/M/2" queue with a threshold. It means that the faster server, which is the primary server at the output link of the queue, remains the same whereas the virtual drop server appears when the number of the packets at the buffer of the queue exceeds a threshold level. In this set of experiments, we use the mathematical model behind the "M/M/2" queue with a threshold, which we have already studied in Section 5.6.

Remember that, the maximum threshold value of the RED algorithm is three times of its minimum threshold parameter unless specifically specified. The other parameters for RED are kept the same as NS's default parameters. Different values of the minimum threshold of both RED and Orange can be applied upon our request. Orange timer (the service time of the unpreferred alternate link) of the bottleneck queue is given in milliseconds and this value is directly proportional of the capacity of the link at the output of the queue of virtual drop server. In Orange, while the packet, which takes service from virtual drop server, is being dropped, the virtual drop server will not consider to drop another packet. For example, 8 ms service time

corresponds to 1 Mbps bandwidth. It means that, if the link is fully utilized, 125 packets will take service per second. In other words, the service time of the virtual drop server is 8 ms/packet.

In this set of simulation, we want to aggregate traffic to generate a total arrival rate "$\lambda = 1200$ pps", and average packet size equal to 100 bytes by aggregating 1000 Poisson sources. The packet size of each source is set by sampling an exponential random variable of average 100 bytes. Each link capacity between the source and the bottleneck node is 100 Mbps with zero link delays. Bottleneck link capacity is Mbps and zero delays. Thus, the average service rate of the link can be computed for 100 bytes packets as 1250 pps. Minimum threshold value of Orange is 14 packets (making c = 15 packets) and service time of the virtual drop server is 5 ms, which corresponds to a service rate of 200 packets per second. It is assumed that there would be a virtual link at the output buffer of the server with a capacity of 200 Kbps.

Using the above parameters, we can calculate the results from the Morrison's equations. Remember that Morrison finds the mean waiting time "W" as;

$$\langle W \rangle = P_0 \left(\frac{\lambda}{\mu_1}\right)^c \left\{ \frac{\mu_1}{(\mu_1-\lambda)^2} \left[(\mu_1/\lambda)^{c-1} - 1\right] + (c-1)\left[\frac{1}{((\mu_1+\mu_2)-\lambda)} - \frac{1}{(\mu_1-\lambda)}\right] + \frac{\lambda}{((\mu_1+\mu_2)-\lambda)^2} \right\} \quad for\, \lambda \neq \mu_1 \qquad \text{Eqn 6.4}$$

This waiting time can be compared with the average queue size in our simulations with a calculation by using the Little's formula. Average queue size (q) is then computed by "$W = q / \lambda$". Using the equation 6.4, waiting time is computed as 0.006989, and correspondingly using the Little's formula, the average queue size is computed as 8.38 packets.

Simulation results can be found in Table 6.2. Average queue size is obtained by simulation as 9.04 packets, which corresponds to the computed value of average queue size 8.38 packets. The difference between the simulation results and the computed values are very close. We are able to state that our derived formulas give the correct results and correspond to the simulation results.

Table 6.2 Simulation results of M/M/2 queue with a threshold

Queue Type	MinTh	Orange Timer (ms)	Sent Packets	Arrival to Router	Arrival to Destination	Average Delay ms	Average Queue Size
Orange	14	5	13178	12680	11728	8.49	9.04

6.3 Topology Bravo with Responsive Sources

To compare our simulation results with theoretical analysis, so far, we have used Poisson sources with an exponential distributed sending times with a mean value, instead of responsive sources. It is obvious that in practical cases, the use of the responsive flows is much higher than the flows with constant sizes. Therefore, we decide to use the responsive sources like TCP, which increase their sending rate (window size) as long as they get acknowledgements from the receiver. It means that responsive flows adjust their sending rates according to the available bandwidth along their path to the final destination. To make this type of experiment, we decide to use the following topology, namely Topology Bravo (see Figure 6.7) with TCP sources, which are responsive to the network condition changes (see Appendix H).

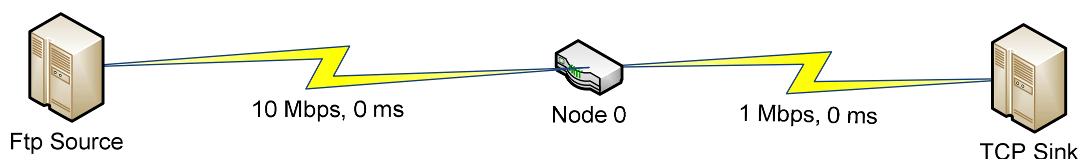

Figure 6.7 Topology Bravo.

In the Topology Bravo, for the sake of simplicity, instead of 1000 Poisson sources, we use an ftp source with packet size of 1000 bytes. The capacity of the link is 10 Mbps with zero delay. FTP packet source is linked to their final destination along a bottleneck link with capacity of 1Mbps with zero delay. We make the link delays zero to prevent their effect to our simulation results. The packet size is fixed at 1000 bytes, and Orange timer is fixed at 10 ms. Maximum buffer size of the queue is fixed at 120 packets as always. Slow Start Threshold is fixed at 500 in NS settings. RED’s maximum threshold value is the three times of the its minimum threshold value in this set of experiments.

First, we made a simulation of ten seconds using Ftp source, which starts to send the packets at the time zero. At the end of ten seconds simulation time, we find the simulation results, which can be found in Table 6.3. Note that, we include “Total Drop” and “Maximum Queue Size” to the simulation results. “Maximum Queue Size” gives the maximum size of the queue during simulation. “Total Drop” gives the number of packets, which have already been dropped by the bottleneck queue during simulation. As it is seen from the simulation results, Orange performs better in average queue size and average packet delay.

Table 6.3 Simulation results of Topology Bravo when ftp source and continuous traffic

Q. Type	Min Th	Arrival to Router	Arrival to Destination	Total Drop	Average Delay ms	T.Put in kbps	Maximum Queue Size	Average Queue Size	Total Download in sec.
DropTail	-	12202	12019	183	768.28	961	120	113.93	-
RED	15	11959	11810	149	157.38	944	120	17.51	-
Orange	10	12192	12019	173	75.54	961	16	7.97	-
Orange	15	12097	11915	182	105.71	959	20	11.57	-

Although, the performance criteria, average packet delay and average queue size even the throughput are enough to compare our proposed algorithm’s performance with that of the other algorithms, we decide to make more realistic analysis to prevent effect of TCP’s unnecessary packet retransmission. To achieve this aim, we add another performance criterion, namely total download time to our simulation results. Even the packets is being retransmitted, simulation stops when the transfer of the complete file of predetermined size finishes. Although, this download time also includes packet retransmissions, we can conclude that queuing algorithm performs better if the download time is smaller.

To realize this, we made another set of simulation with an Ftp source, which produces only 10,000 packets to transfer to their final destination. We use the “produce” command in NS to produce the ftp traffic of predetermined size like in the following code.

```
$ns at 0 "$ftp0 produce 10000"
```

Total size of the file that needs to be transferred comes to 10 MB. Simulation finishes when the download of the file finishes. The results of the simulation can be found in Table 6.4. It is obviously observed that the total download times decreases when Orange algorithm is applied. Previous performance parameters; throughput, average delay, and average queue size are still better when Orange algorithm is used.

Table 6.4 Simulation results of Topology Bravo when ftp source and download of a file

Q. Type	Min Th	Arrival to Router	Arrival to Destination	Total Drop	Average Delay ms	T.Put in kbps	Maximum Queue Size	Average Queue Size	Total Download in sec.
DropTail	-	10245	10123	122	736.64	961	120	87.44	84.22
RED	15	10214	10052	162	89.71	930	59	9.36	86.42
Orange	10	10154	10016	138	74.55	961	14	7.85	83.34
Orange	15	10098	10018	80	105.56	959	20	11.55	83.54

6.4 Topology Charlie and More on Testing the Download Performance

After we show that, Orange performs better than DropTail and RED algorithms in throughput and average queuing delays, we decide to extend our simulation experiments using the download time criteria for a topology with two different TCP sources, namely Topology Charlie that can be found in Figure 6.8 (see Appendix I). In the following topology, there are two ftp sources that produce 10,000 packets that need to be transferred to their destinations over the same bottleneck link. When both of the sources complete to transfer their packets, the simulation finishes. The time of which a source completes transmission is the download time for that source. The sum of the download times of these two sources is called the total download time, which are our performance criteria as well as the average queuing delays in this set of experiments. As the total number of the packets that are transferred is 20,000 and the packet size is 1000 bytes, we can say that total size of the file to be downloaded is 20 MB. Orange timer is fixed at 6.5 ms, and RED's maximum threshold value is the three times of the its minimum threshold value in this set of experiments. Maximum

buffer size of the queue is 120 packets. The simulation results can be found in Table 6.5.

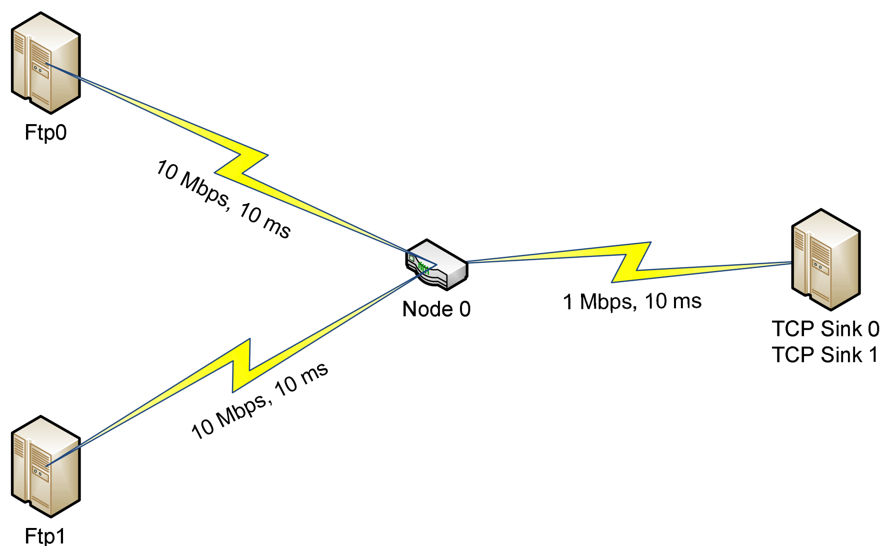

Figure 6.8 Topology Charlie.

Table 6.5 Simulation results of Topology Charlie

Q. Type	minTh	Arrival to Router	Arrival to Destination	Total Drop	Average Delay ms	Maximum Queue Size	Average Queue Size	Flow 1 Download in sec.	Flow 2 Download in sec.	Total Download in sec.
DropTail	-	20029	20011	28	719.40	120	81.96	157.32	168.56	325.88
RED	10	20329	20057	272	131.74	98	12.23	152.44	168.07	320.51
Orange	10	20407	20025	382	79.04	12	5.90	148.68	168.94	317.62
Orange	15	20301	20031	270	119.15	16	10.63	148.67	169.36	318.03
Orange	20	20193	20011	182	144.07	21	13.56	145.96	169.42	315.38

As the average throughput does not correspond to the goodput, which we have described earlier, and include the unnecessary packet retransmissions, we prefer to use total download parameter to be sure about the best transfer time of a file at a predetermined size. One can easily see that when Orange algorithm is used, the average queue size, and average delay in ms (milliseconds) significantly decreases compared to DropTail, and RED. Moreover, when Orange algorithm is used the total download times are shorter compared to that of DropTail, and RED. This means that Orange algorithm prevents unnecessary packets retransmissions and performs better than other active queue management algorithms.

6.5 Topology Delta for Orange's Performance Tests

To continue to test more the Orange's performance over similar algorithms we decide to use the Topology Delta (see Figure 6.9), where there are three different traffic sources competing for one common destination through one common link. We have developed a Tcl script that creates the nodes, links, and traffic sources (see Appendix J).

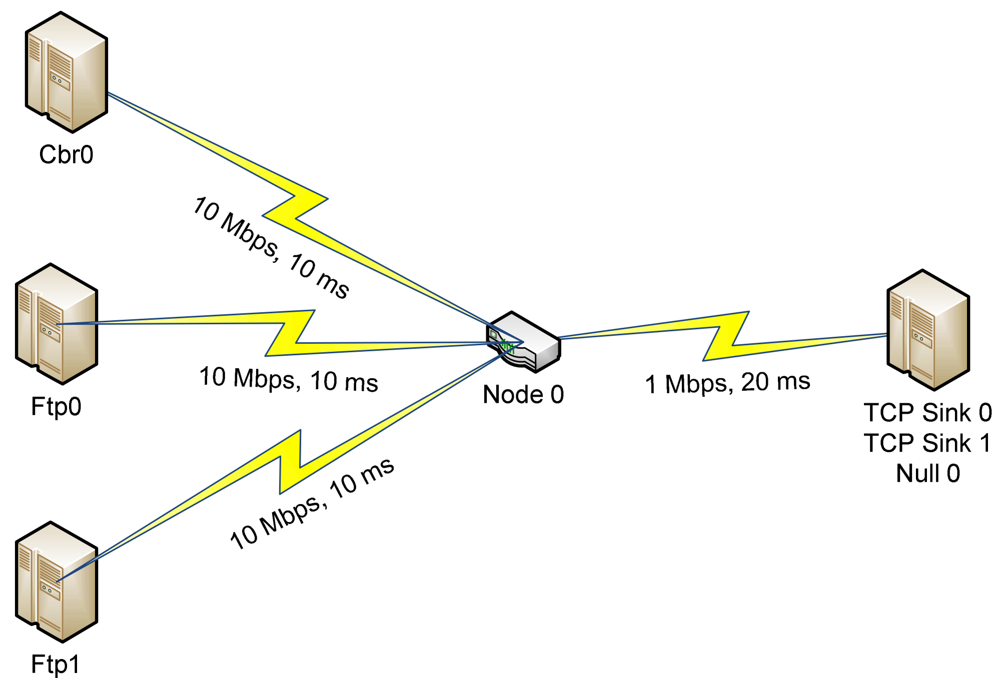

Figure 6.9 Topology Delta.

The physical buffer size of the bottleneck queue is fixed at 120 packets based on published rules of thumb for accommodating the network bandwidth delay product. Queue size is calculated in packets. TCP packet sizes are 1000 bytes. CBR packet sizes are 1000 bytes. Simulation time is 5 seconds. Maximum threshold value of the RED algorithm is three times of its minimum threshold parameter unless specifically specified. The other parameters for RED are the NS's default parameters. The orange timer (the service time of the unpreferred alternate link) of the bottleneck queue is 6.5 ms. Simulation results can be found in the Table 6.6.

Table 6.6 The results of the simulation of Topology Delta

MinTh	Queue Type	Sent Packets	Arrival to Router	Arrival to Destination	Average Throughput Kbps	Average Delay ms	Average Queue Size	Total Drop by Router	Early Drop	Force Drop
	Drop Tail	971	971	601	973.64	730.82	93.90	284	0	284
10	Red	1026	1026	596	975.91	363.30	38.66	429	429	0
10	Orange	935	935	583	980.70	115.24	8.90	351	351	0
15	Red	1014	1014	603	976.69	485.91	56.94	376	342	34
15	Orange	954	954	587	980.04	153.27	13.42	366	366	0
20	Red	1029	1029	602	976.33	595.96	72.44	371	300	71
20	Orange	956	956	592	979.82	194.97	18.47	363	363	0

Note that we include "Total Drop by Router", "Early Drop", and "Force Drop" parameters to the simulation results. "Early Drop" gives the number of the packets that have already been dropped by an active queue management algorithm when the (average) queue length exceeds a threshold level. "Force Drop" gives the number of packets that have been already dropped by the FIFO queue because there is no physically space left in the queue buffer. "Total Drop by Router" gives the sum of "Force Drop" and "Early Drop" parameters. We have seen from the results of our experiments that when Orange algorithm is used, we achieve higher throughput and lower queuing delays compared to drop tail gateway and RED algorithms.

6.6 Topology Echo and Main Experimental Work

Up to now, we have used topologies, which could be best suited to experimental aims. However, in most of the practical cases, most of the traffic is formed by the responsive sources of large amounts. Those are the flows of surfing a web site, or download a file from the internet. It is more complicated to control those flows. In order to test our algorithm's performance in a topology, which we can see in most of the practical cases, we decide to use a sample topology, namely Topology Echo, consisting of heterogeneous TCP flows whose link delays are varying. This topology has also been studied by Kinicki and Zheng (Kinicki and Zheng, 2001). They have

used this topology to test their own algorithm's performance with that of RED algorithm. They claim that the chosen RED parameters in their work give the best result when RED algorithm is applied. To test our algorithm with other IP level congestion control methods, we have chosen the same topology, which has many heterogeneous TCP Reno flows, and this topology is best suited for our performance comparisons of our proposed algorithm.

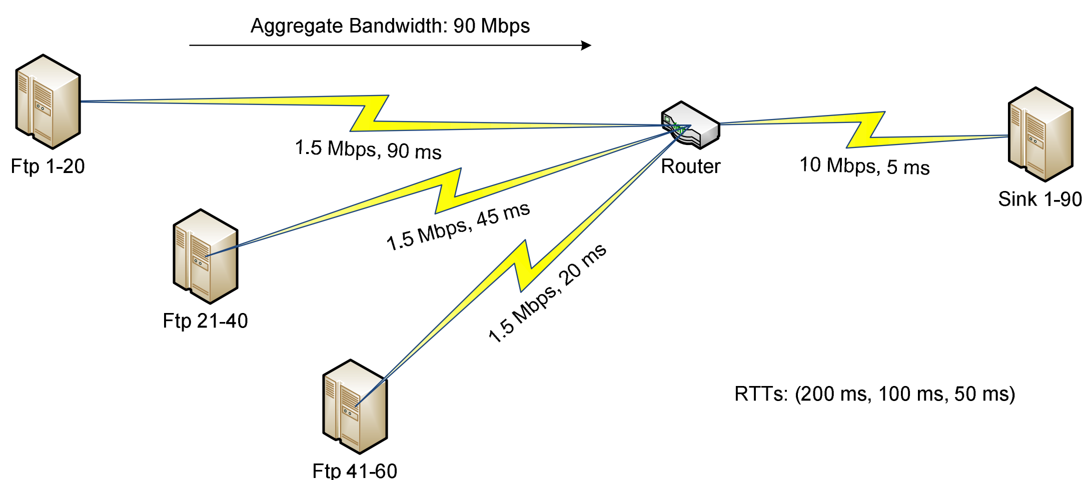

Figure 6.10 Topology Echo.

In the Topology Echo, which is given in Figure 6.10, all flows are divided into three flow groups (fragile, average, and robust) based on the instantaneous round trip time of each flow. The mentioned Orange router maintains a single flow queue for each flow group, which is a FIFO queue that stores a pointer to a packet in the router queue for each packet. The aim of this topology consisting of three flow groups is to establish a real network situation, which has many flows with too many RTT's.

In this work, we run a series of simulation experiments using the NS simulator to compare the performance of Orange with RED and its variants with heterogeneous TCP Reno sources. The simulated network topology consists of one router, one sink and a number of simulated FTP sources. Each FTP source feeds 1000-byte packets into a single congested link attached to the router. The TCP ACK packets are 40

bytes long and each source has a window size of 64 packets. The capacity of the bottleneck link is 10 Mbps with a 5 ms delay to the sink. When the demand is kept constant, the number of the flows that generates the demand has a negative effect on performance. We choose one-way link delays for the fragile, average and robust sources of 95 ms, 45 ms and 20 ms respectively. Thus, the fragile, average and robust flows have round trip times of 200 ms, 100 ms and 50 ms when there is no queuing delay at the router. The router queue size was fixed at 120 packets based on published rules of thumb for accommodating the network bandwidth delay product. All simulations for this study run for 100 simulated seconds and include an equal number of fragile, average and robust TCP flows. Half of the flows in each flow group start at time zero the second half start at time 2 seconds. For example, for a 60-flow simulation, 10 fragile, 10 average and 10 robust flows start at time 0, and the remaining 30 flows start at 2 seconds. The first 20 seconds of simulated time are not considered to reduce the startup and transient effects. The sum of the capacities of all the incoming flows is held constant at 90 Mbps for all simulations in this study regardless of the number of flows. Thus when the number of flows are increased the individual link capacities are proportionally decreased. Unless specifically specified the values for RED parameters of minth and maxth are set in such a way that maxth is three times of the minth.

We have developed a Tcl script to construct the Topology Echo in the simulation (see Appendix K). The number of flow groups, the number of the flows in a group and the aggregate bandwidth demanded are the parameters to this script.

In this set of experiments, our aim is to keep the aggregate throughput high but the average packet delay and the average queue sizes low. We have four sets of simulation; each has a different minimum threshold value (10, 15, 20, 25). In RED queue type, when the average queue size exceeds the minimum threshold value, queue starts to drop the incoming packets according with a dropping probability value based on the calculation of the maximum dropping probability and the value of the average queue size. In Orange queue type, when the current queue size exceeds the minimum threshold value, queue starts to drop the incoming packets according the busy – idle status of the alternate drop server. RED has a maximum threshold

value parameter to drop all the incoming packets when the average queue size exceeds it. Maximum threshold of the RED is three times of its minimum threshold parameter and unless it is specifically specified. Specifically specified RED's maximum threshold values in the simulation are indicated in tables.

On the other hand, Orange has no maximum threshold parameter, but it has the parameter, which is the service time of the alternate server. It is the busy period between the time that the Orange drops a packets and the time that the Orange queue will consider another packet to drop (busy time for dropping a packet). This time value (Orange Timer) is not constant, it is exponentially distributed about a mean average value, which is parameter of Orange queue type. We have simulated our sample topology with the values of the service time of the alternate server from values of 1 second to 10000 seconds in order to test the effect of the service time to the Orange's overall performance. As the Orange's service time goes to infinity, its operating behavior approaches the drop tail. With a big service time, Orange drops a packet and after this time, it never drops any packets because its alternate drop server is busy during the simulation time. That is why drop tail queue type is not included in our simulation. The results of the experiments are given in Tables 6.7, Table 6.8, Table 6.9, and Table 6.10. Average throughput graph when threshold is 10 and average delay graph when threshold is 10 are also given in Figure 6.11, and Figure 6.12.

Table 6.7 Simulation results of Topology Echo when threshold is 10

Orange Timer	Arrival to Router	Arrival to Destination	Average Throughput Kbps	Average Delay ms	Average Queue Size	Total Drop by Router	Early Drop	Force Drop
RED-30	105809	96107	9610.84	79.18	23.82	9745	9745	0
1	105795	94927	9493.04	62.30	7.00	10872	10872	0
2	106217	95164	9516.54	65.09	7.99	11053	11053	0
3	105929	95227	9522.86	68.87	9.35	10697	10697	0
4	107522	95915	9591.63	70.94	11.19	11605	11605	0
5	106453	95971	9597.22	69.56	14.59	10474	10474	0
6	106022	96067	9606.77	74.24	17.06	9950	9950	0
6.5	106092	96122	9612.53	77.86	18.71	9977	9977	0
7	105890	96145	9614.52	84.02	23.24	9740	9740	0
8	105538	96153	9615.43	119.80	63.99	9369	9369	0
9	104507	96154	9615.44	97.18	39.18	8333	8333	0
10	103540	96154	9615.53	105.42	54.74	7401	7401	0
50	101777	96154	9615.53	146.67	110.95	5606	1551	4055
100	102084	96154	9615.52	146.87	112.44	5908	782	5126
1000	102427	96154	9615.54	145.45	111.70	6278	80	6198
10000	102450	96154	9615.53	146.49	113.29	6301	8	6293

Table 6.8 Simulation results of Topology Echo when threshold is 15

Orange Timer	Arrival to Router	Arrival to Destination	Average Throughput Kbps	Average Delay ms	Average Queue Size	Total Drop by Router	Early Drop	Force Drop
RED-45	105047	96153	9615.48	88.27	33.83	8908	8908	0
RED-30	105602	96059	9606.00	80.66	25.26	9530	9530	0
1	106308	95465	9546.53	65.65	11.23	10849	10849	0
2	106043	95523	9552.46	70.38	11.97	10514	10514	0
3	105891	95612	9661.40	73.50	13.56	10287	10287	0
4	107332	96037	9603.82	74.87	16.43	11299	11299	0
5	106098	96100	9610.26	73.04	18.98	10014	10014	0
6	105689	96133	9613.37	77.51	21.00	9564	9564	0
6.5	105709	96133	9613.41	81.04	22.37	9592	9592	0
7	105770	96154	9615.63	88.56	28.26	9602	9602	0
8	105517	96154	9615.53	119.08	93.88	9364	9364	0
9	104463	96154	9615.64	97.96	40.56	8327	8327	0
10	103575	96153	9615.44	105.41	54.89	7402	7402	0
50	101743	96154	9615.43	146.65	110.70	5577	1559	4018
100	101980	96154	9615.53	145.73	111.26	5846	782	5064
1000	102207	96154	9615.63	146.94	113.26	6052	78	5974
10000	102529	96154	9615.43	146.01	112.75	6345	8	6337

Table 6.9 Simulation results of Topology Echo when threshold is 20

Orange Timer	Arrival to Router	Arrival to Destination	Average Throughput Kbps	Average Delay ms	Average Queue Size	Total Drop by Router	Early Drop	Force Drop
RED-60	104402	96154	9615.53	96.67	43.32	8275	8275	0
RED-30	105302	95915	9591.64	82.41	26.99	9401	9401	0
1	105999	95608	9560.86	69.41	15.02	10392	10392	0
2	105929	95616	9561.77	74.39	16.12	10318	10318	0
3	105827	95857	9585.87	78.04	17.99	9965	9965	0
3.5	106326	96086	9608.79	75.12	19.77	10240	10240	0
4	106818	96110	9611.11	78.83	20.83	10693	10693	0
4.5	105367	96078	9607.86	81.51	21.81	9289	9289	0
5	105779	96111	9611.19	77.17	23.95	9663	9663	0
5.5	105220	96128	9612.82	82.74	23.57	10240	10240	0
6	105601	96152	9615.28	81.98	26.22	9439	9439	0
6.5	105278	96147	9614.90	85.78	27.43	9156	9156	0
7	105529	96153	9615.35	89.93	30.75	9383	9383	0
8	105570	96154	9615.53	118.80	62.90	9373	9373	0
9	104455	96154	9615.54	99.91	42.76	8302	8302	0
10	103542	96154	9615.53	105.76	54.73	7402	7402	0
50	101735	96154	9615.73	146.35	110.72	5587	1548	4039
100	102009	96154	9615.53	146.56	112.02	5860	784	5076
1000	102252	96154	9615.53	145.96	112.78	6073	80	5993
10000	102406	96154	9615.63	146.43	113.28	6263	8	6255

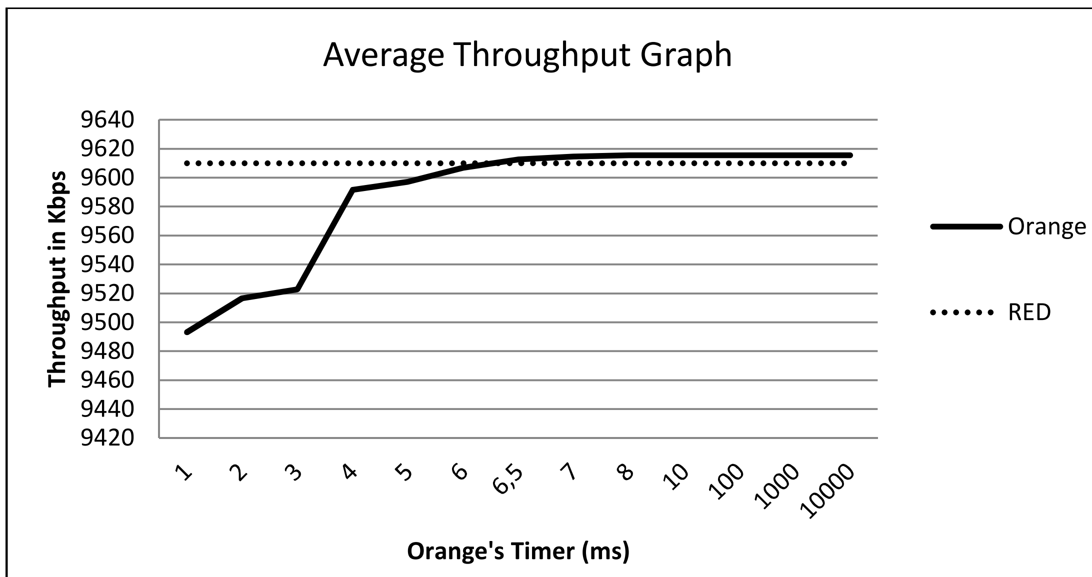

Figure 6.11 Average throughput graph when threshold is 10.

Table 6.10 Simulation results of Topology Echo when threshold is 25

Orange Timer	Arrival to Router	Arrival to Destination	Average Throughput Kbps	Average Delay ms	Average Queue Size	Total Drop by Router	Early Drop	Force Drop
RED-75	103970	96154	9615.54	103.78	52.56	7819	7819	0
RED-30	104466	94820	9482.20	83.27	28.80	9658	9658	0
1	105803	95829	9583.09	72.12	19.23	9970	9970	0
2	105878	95925	9592.58	77.26	20.23	9955	9955	0
3	105613	95989	9599.21	81.59	22.24	9620	9620	0
3.5	106092	96154	9615.48	78.84	24.36	9938	9938	0
4	106433	96128	9612.87	82.87	25.60	10319	10319	0
4.5	105154	96101	9610.27	84.07	25.62	9053	9053	0
5	105438	96152	9615.27	81.30	28.38	9289	9289	0
5.5	104946	96141	9614.22	86.47	27.58	8805	8805	0
6	105104	96154	9615.44	85.95	31.01	8947	8947	0
6.5	105055	96154	9615.54	89.07	31.65	8919	8919	0
7	105172	96154	9615.44	93.13	34.45	9015	9015	0
8	105536	96154	9615.63	121.48	66.15	9358	9358	0
9	104329	96154	9615.54	99.75	42.92	8176	8176	0
10	103548	96154	9615.53	105.02	54.56	7398	7398	0
50	101750	96154	9615.53	146.35	110.79	5596	1543	4053
100	102001	96153	9615.45	146.81	112.51	5848	783	5065
1000	102365	96154	9615.63	145.82	112.54	6212	79	6133
10000	102211	96154	9615.49	146.22	113.00	6072	8	6064

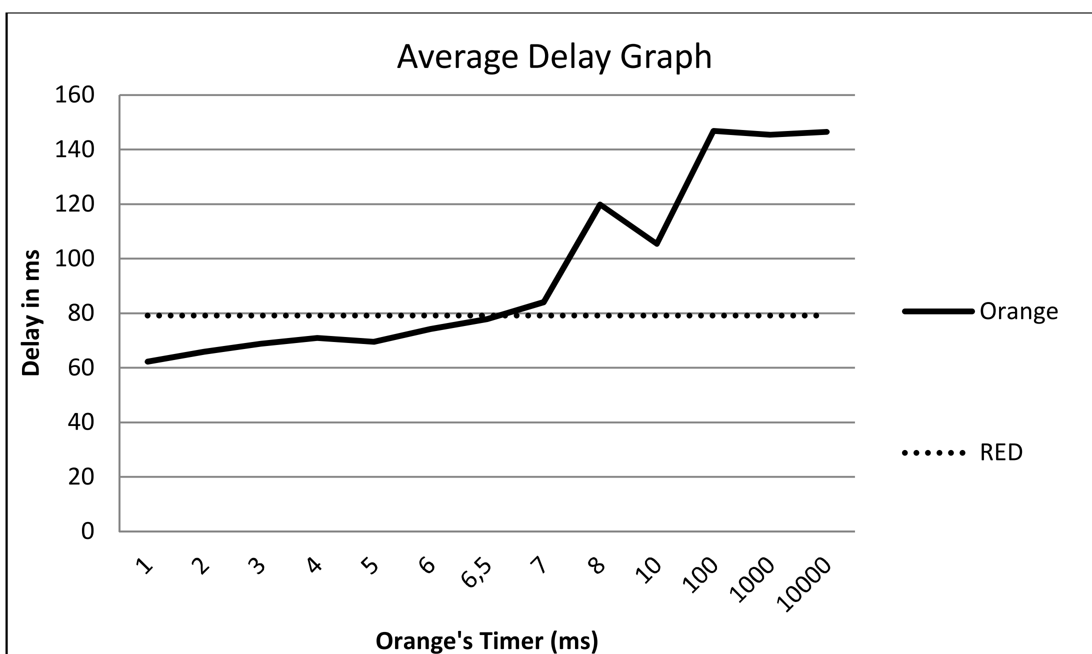

Figure 6.12 Average delay graph when threshold is 10.

6.7 Analysis of the Simulation Results

The performance parameters that we have compared RED and Orange are the average throughput (Kbps), average delay (ms), average queue size. It is obvious that in most of the regions, Orange has better performance compared to RED and Drop Tail especially when the service time of the alternate server is around from 4 ms to 7 ms. Orange provides better performance for smaller timer values as the minimum threshold value increases.

In Table 6.7 when the threshold is 10, RED's throughput is measured as 9610 Kbps, average delay, and average queue are measured as 79.18, and 23.82, respectively. In this set of experiments, Orange's minimum threshold value is fixed at 10 packets. It means that Orange starts to drop the incoming packets when the queue size exceeds 10 packets. Orange's service time is adjusted from low values to the high values. When it gets higher, Orange approaches to work like a DropTail queue. Orange drops the packet, and it never gets idle because the service time for that packet is too high to consider another packet to drop or not. While keeping the threshold at a fixed level which is 10 for this set of simulations, total throughput increases, as the service time increases whereas average delay, and average queue size decrease. Orange gives better results than RED when Orange's timer is adjusted around 6.5-7. This is the point where Orange provides higher throughput values and lower delay values than that of RED. It is obvious that average delay is directly proportional to average queue size. It increases as the average queue size increases.

In Table 6.8 when the threshold is 15, RED's throughput is measured as 9615 Kbps, average delay, and average queue are measured as 88.26, and 33.83, respectively when RED's minimum threshold value, and maximum threshold value are fixed at 15, 45 respectively. RED's throughput, average delay, and average queue are measured as 9606, 80.65, 25.26, respectively when RED's minimum threshold value, and maximum threshold value are fixed at 15, 30 respectively. In this set of experiments, Orange's minimum threshold value is fixed at 15 packets. As we know, RED starts to consider dropping packets when the average queue size exceeds its minimum threshold value. Therefore, as we expect, the average delay and average

queue size are a little bit more than the previous results. Orange gives better results -higher throughput and lower delay- than RED when Orange's timer is adjusted around 5-6 ms.

In Table 6.9 when the threshold is 20, RED's throughput is measured as 9615 Kbps, average delay, and average queue are measured as 96.67, and 43.31, respectively when RED's minimum threshold value, and maximum threshold value are fixed at 20, 60 respectively. RED's throughput, average delay, and average queue are measured as 9591, 82.40, 26.98, respectively when RED's minimum threshold value, and maximum threshold value are fixed at 20, 30 respectively. Orange's minimum threshold value is fixed at 20 packets. Orange gives better results -higher throughput and lower delay- than RED when Orange's timer is adjusted around 4-5 ms.

In Table 6.10 when the threshold is 25, RED's throughput is measured as 9615 Kbps, average delay, and average queue are measured as 103.77, and 52.56, respectively when RED's minimum threshold value, and maximum threshold value are fixed at 25, 75 respectively. RED's throughput, average delay, and average queue are measured as 9482, 83.27, 28.80, respectively when RED's minimum threshold value, and maximum threshold value are fixed at 25, 30 respectively. Orange's minimum threshold value is fixed at 25 packets. Orange gives better results -higher throughput and lower delay- than RED when Orange's timer is adjusted around 4 ms.

When we try to track the change the change in Orange's timer optimum value as compared to the change in the set threshold value, we can fit an inverse proportional relation to the square root of threshold (K). For instance, if we compare the simulation results where the threshold value is 10 with the results where the threshold value is 25, service time of the alternate server should be multiplied by $\sqrt{10/25} = \sqrt{0.4} \cong 0.632$. Thus, to get the optimum value of the alternate server's average service time, if we multiply best service time value where the threshold is 10 with this coefficient, we can easily see that the result fits very well with the result where the threshold is 25. (6.5 ms * 0.632 = 4.10 ms). This last value is the best service time value of the alternate server where the threshold is 25.

Consequently, empirically fitting relationship can be formulated as

$$Service\ Time \propto \frac{1}{\sqrt{K}} \qquad \text{Eqn 6.5}$$

where “K” is Orange’s threshold value for the best performance of our simulation.

Hence, we can state that, from the analysis of the simulation, empirical results suggests with our used simulation parameters are $Service\ Time = \frac{20}{\sqrt{K}}$ in milliseconds. A comparison between the simulation results and this empirical formula is given in Table 6.11.We can easily see that the results fit well.

Table 6.11 A comparison between simulation results and empirical formula

Threshold Applied	Orange's Timer in ms	
	Best Result from Simulation	Calculation from Empirical Formula
10	6.50	6.32
15	6.00	5.16
20	4.50	4.47
25	4.00	4.00

6.8 Empirical Validation of Orange’s System Parameters

We have made our experiments for different threshold values and different service times for slower server in order to find the best operating point of our algorithm in a congested network environment, which includes responsive flows. Our aim is to find a relation between the values of the threshold and the service time of the slower server at the operating point from the experiments and the mathematical analysis. Şiş (1994) studied the optimum threshold value of an M/M/2 queue where Poisson arrivals, and exponentially distributed service times are of interest (when the service rates of both servers are predetermined). He proved that the first order approximate value of optimum threshold, is the largest non-negative integer which satisfies (if there is no such non-negative integer, it is zero)

$$\tilde{K} \leq \frac{\mu_1 - \lambda}{\mu_2} - 1 \qquad \text{Eqn 6.6}$$

This approximate value for the optimum value of the threshold gives satisfactory result under the assumption that "μ_1" is considerably greater than "μ_2" and "$\mu_1 >> \lambda$". Here, the results are approached as there is a continuous flow of traffic arriving to the queue with the average rate of "λ" units/time and similarly μ_i units/time is the continuous average out-flow through link i. Therefore, the results are valid in the systems where memory-less sources like Poisson sources are applied. If "μ_1" is not considerably larger than "μ_2", it is clear that threshold is nearly zero. When "μ_1" is considerably larger than "μ_2", if "$\mu_1 >> \lambda$", for optimum threshold we can use the approximate value in Equation 6.6. The only remaining case is the case where "$\mu_1 >> \mu_2$", but "$(\mu_1-\lambda) \approx 0$". There, actually a non-zero threshold value occurs which is not anticipated in our approximation. Although this afore mentioned analysis can be made to find the expected delay value to relate it with the threshold value, this would be restricted to the case where Poisson arrivals and exponentially distributed service times are involved.

In order to test our algorithm's performance in a network where the responsive flows are dominant, we use the responsive sources (ftp sources) in our simulation. Responsive sources probe the available bandwidth in the network, and they adjust their sending rate as long as there is no packet loss. Arrival rate will be almost the same as the service rate of the server. We can easily say that, in our experiment "$(\mu_1-\lambda) \approx 0$". We need to find an equation for this case in terms of μ_1, μ_2, λ, and K under these circumstances where responsive flows are involved.

Padhye and his friends (Padhye *et al.,* 1998) develop a simple analytic characterization of the steady state throughput of a bulk transfer TCP flow (i.e., a flow with a large amount of data to send, such as FTP transfers) as a function of loss rate and round trip time. Their model captures not only the behavior of TCP's fast retransmit mechanism but also the effect of TCP's timeout mechanism on throughput.

In their work, N_t represents the number of packets transmitted in the interval [0,t] and "B_t (N_t/t)" represents the throughput on that interval. Thus, B_t represents the throughput of the connection, rather than its goodput. They define the long-term steady-state TCP throughput B to

$$B = \lim_{t\to\infty} B_t = \lim_{t\to\infty} \frac{N_t}{t} \qquad \text{Eqn 6.7}$$

They have assumed that if a packet is lost in a round, all remaining packets transmitted until the end of the round are also lost. Therefore they define p to be the probability that a packet is lost, given that either it is the first packet in its round or the preceding packet in its round is not lost. They are interested in establishing a relationship B(p) between the throughput of the TCP connection and the loss probability (p).

In their work, when timeout occurrences are ignored, B(p) is derived to be;

$$B(p) \approx \frac{1}{RTT}\sqrt{\frac{3}{2bp}} \qquad \text{Eqn 6.8}$$

where "b" is the number of packets acknowledged by a received ACK. In many TCP implementations, "b = 2". When timeouts are taken into account, they derive the B(p) as;

$$B(p) \approx \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0 min\left(1, 3\sqrt{\frac{3bp}{8}}\right) p(1+32p^2)} \qquad \text{Eqn 6.9}$$

By this formula, we can easily observe that TCP favors the flows with short RTT. It means that when downloading a file from a closer server, the download performances will be better. We can observe that the relationship between loss rate p and throughput is not linear but an inverse square root relation! It means when p is increased 4 times, throughput drops to half.

As we have already shown from the simulation results in section 6.7, while the service time of the drop server increases, the optimum value of the threshold decreases in order to achieve the best operating point. If the service time of the drop server were too low, the threshold would be high enough to prevent unnecessary packet drops. If the threshold were too low, we need high values of the service time of the drop server to make the drop server idle after dropping a packet. To use the drop server for enough times, it must work faster. Therefore, we can easily say that

the optimum value of the threshold is inversely proportional to the service time of the drop server. We have found empirically a relation like;

$$Service\ Time = \frac{1}{\mu_2} \propto \frac{1}{\sqrt{K}} \qquad \text{Eqn 6.10}$$

On the other hand, according to the Padhye and his friends (Padhye *et al.,* 1998), we can state that TCP's throughput is inversely proportional with the square root of dropping probability (P). We can also intuitively claim that the dropping probability is inversely proportional with the threshold (K):

$$p \propto \frac{1}{K} \qquad \text{Eqn 6.11}$$

If we think alternate server as a real server, departures from it contributes to the total throughput. Therefore, the service rate of the alternate server, and the throughput can be assumed that they are directly proportional.

$$\mu_2 \propto B(p) \qquad \text{Eqn 6.12}$$

If we use the last two formulas in Padhye's simple throughput equation, then we get

$$B(p) \propto \frac{1}{RTT}\sqrt{\frac{3}{2bp}} \approx \frac{1}{RTT}\sqrt{\frac{3K}{2b}} \qquad \text{Eqn 6.13}$$

In general TCP implementations, b value is fixed as 2 and we can assumed that the RTT is constant during simulation so without specifying proportionality constants, we can end up with

$$\mu_2 \propto \sqrt{K} \qquad \text{Eqn 6.14}$$

or, alternate server's service time is inversely proportional with square root of threshold (K):

$$Orange's Timer = \frac{1}{\mu_2} \propto \frac{1}{\sqrt{K}} \qquad \text{Eqn 6.15}$$

We can just conclude that, empirically the best value of the service time is inversely proportional with the square root of the threshold value applied as we have already stated in Equation 6.5.

Furthermore, if we can estimate as "a rule of thumb", the best value of μ_2 departing from the capacity of the main (bottleneck) link, it can be estimated in the order of "1/10" to "1/5" of the capacity of the main link. And by using our empirical formula, we can try to find an optimal threshold (K) value. This finishes selecting Orange parameters that gives the best operating point. This parameter estimation procedure is much simpler, effective, and more meaningful than the tuning the complex RED parameters.

The other way around, if we take the threshold value of Orange departing from the minimum threshold value proposed for RED implementers, we can easily calculate the best optimum value of Orange's timer for the best performance of active queue management.

Explanation of this relates the drop server to behave like a TCP friendly source. The implication of this can be very meaningful. Mentioning TCP friendliness in general means reacting to congestion in the same way as TCP, considering only Triple Duplicate (TD) packet loss occurrences that result in TD, this would mean to be conformant with the throughput Equation 6.8. We have demonstrated that our empirical result is in accordance with Equation 6.8, therefore suggesting the TCP friendliness for the best operating conditions.

However, keeping in mind that alternate server's output is, in return as retransmission, a load for the original sender (TCP source), they will be part of the offered load, hence throughput is in relation with alternate server's link capacity or service time.

CHAPTER SEVEN
CONCLUSIONS

7.1 Drawbacks of Current Active Queue Management Algorithms

The network calculation on detecting incipient congestion is called active queue management (AQM) as we describe earlier. Many active queue management schemes studied in recent literatures are based on early congestion notification to sources. These schemes are classified into AQM schemes based on queue value. This category is also called as queue-based AQM algorithm, which uses the average (or instantaneous) queue length to calculate the packet dropping probability. RED is the most widely used active queue management algorithm, which is the recommendation approach by Internet Engineering Task Force (IETF).

However, although RED is certainly an improvement over traditional drop-tail queues, it has several drawbacks in practical usages. Abbosov & Korukoglu (2009) summarizes the drawbacks of RED algorithm as follows;

> RED performance is highly sensitive to its parameter settings. In RED, at least 4 parameters, namely, maximum threshold (max_{th}), minimum threshold (min_{th}), maximum packet dropping probability (max_p), and weighting factor (w_q) have to be properly set. RED performance is sensitive to the number of competing sources/flows. RED performance is sensitive to the packet size. With RED, wild queue oscillation is observed when the traffic load changes (Abbosov & Korukoglu, 2009).

In addition, one of RED's main weaknesses is usage of indirect congestion indicator namely the average queue length. The misbehavior of packet dropping or marking occurs when the queue gets large but exhausting rapidly or small but filling up rapidly. This phenomenon is called as lag domino effect. While the existence of a persistent queue indicates congestion, its length gives very little information to the severity of congestion. A single source which is unaware the congestion situation and

transmitting at a rate greater than the bottleneck link capacity can cause a queue to build up easily as a large number of sources can do.

Other weakness of RED is the parameter configuration problem. Since the RED algorithm relies on queue lengths, it has an inherent problem in determining the severity of congestion. As a result, RED requires a wide range of parameters to operate correctly under different congestion scenarios. While RED can achieve an ideal operating point, it can only do so when it has a sufficient amount of buffer space and is correctly parameterized. The average queue size varies with the level of congestion and with the parameter settings and the throughput is sensitive to the traffic load and to RED parameters. Therefore, tuning RED parameters is unavoidable, especially under realistic environment.

7.2 Advantages of Orange

The most effective detection of congestion occurs at the node level. The router at the nodes can reliably distinguish between propagation delay and persistent queuing delay. The router has a cohesive view of the queuing behavior over time; the perspective of individual connections is limited by the packet arrival patterns for those connections. In addition, a router at the nodes is shared by many active connections with a wide range of round trip times, tolerances of delay, throughput requirements, etc.; decisions about the duration and magnitude of transient congestion to be allowed at the node are best made by the router at the node itself in IP level.

Orange as an IP level active queue management algorithm, which can be applied at the current Internet routers can be useful in controlling the average queue size even in a network where the transport protocol cannot be trusted to be cooperative. Orange can control the improving congestion and provides the upper bound on the buffers. Orange gives best performance for a network where the transport protocol responds to congestion indications from the network. Orange is designed for a network where a single marked or dropped packet is sufficient to signal the presence

of congestion to the transport-layer protocol. The probability that the Orange chooses a particular connection to notify during congestion is roughly proportional to that connection's share of the bandwidth at the gateway.

Orange is an effective mechanism for congestion avoidance at the routers, in cooperation with network transport protocols. If Orange drops packets when the queue size exceeds the threshold, rather than simply setting a bit in packet headers, then Orange controls the average queue size in effect. This action provides an upper bound on the average delay at the router. For Orange, the rate at which the algorithm marks packets for dropping, depends on service time of the alternate server (unpreferred link). This approach avoids the global synchronization that results from many connections decreasing their windows at the same time. Because, Orange's timer is set randomly and two successive packets may or may not be dropped. Orange gateway is a simple router algorithm that could be implemented in current networks or in high-speed networks of the future. Orange allows practical design decisions to be made about the average queue size and the maximum queue size allowed at the router.

Another advantage of Orange is that it has less adjustable parameters than RED. Orange requires minimum amount of processor and memory consumption. Tuning and optimizing the parameters used for generic topologies has easier in Orange rather than RED.

In addition, it avoids the global synchronization, which results from both TCP connections reducing their windows at the same time where each connection goes through Slow-Start, reducing the window to one, in response to a dropped packet, and thus results a lower throughput and higher queuing delays.

7.3 Concluding Remarks

The main contribution of this work is to present an IP level congestion control mechanism to control the performance of a traffic network at the node level. In this work, a new active queue management algorithm called Orange is designed and evaluated. The main idea behind Orange comes from the analysis of two heterogeneous servers and one queue with a threshold-based queuing system in order to achieve both higher throughput and lower queuing delays. In addition, we consider to find out an empirical relationship between the system parameters of our algorithm using the mathematical analysis. Simulation results are used to tune up the empirical formulation. By achieving this aim, we consider to use a virtual drop server to drop the incoming packets when the actual queue size exceeds a threshold level. The only adjustable parameter based on the changing conditions of the network is the service time of the virtual drop server. Since for many applications, this service time is not usable, we consider it an important and distinguishing characteristic of our work.

By using the threshold type policy and the use of virtual drop server, we have proposed a new approach to drop or mark packets when the congestion will likely occur. The primary performance parameter is the mean number of customers in the system, and accordingly the average waiting time per packet as well as the throughput of the network.

This study confirms that generally Orange performs better than RED due to the fact from simulations that it results in higher throughput values and lower queuing delays (thus the lower mean waiting times per packet) for the networks with heterogeneous flows. Orange simulations indicate that Orange requires less parameter settings than RED.

We can propose that Orange will replace RED as an active queue management algorithm to decide which packets are to be marked to indicate a congestion condition for the current Internet routers. We still chose to drop them to warn TCP friendly sources (responsive or adaptive) against that possible congestion situation.

7.4 Recommendations for future research

Policy for using an alternate route (link) for the packets, which are destined to a certain network, may be implemented as a threshold checking on the size of this flows dedicated queue's size. In addition, RED and its derivatives use threshold on average queue size in order to decide whether to drop or accept an arriving packet to the node. Our analysis would be extended to find new policies, which could be more efficient in that extent.

There are many areas for further research on Orange gateways. The foremost open question involves determining the optimum average queue size for maximizing throughput and minimizing delay for various network configurations. The answer of this this question is heavily dependent of the characterization (modeling) of the network traffic and behavior of traffic offering servers.

One area for further research concerns traffic dynamics with a mix of Drop Tail and Orange gateways, as would result from partial deployment of Orange gateways in the current internet. Another area for further research concerns the behavior of the Orange gateway machinery with transport protocols other than TCP, including open or closed-loop rate-based protocols.

The list of packets marked by the Orange could be used by the gateway to identify connections that are receiving a large fraction of the bandwidth through the gateway. The gateway could use this information to give such connections lower priority at the gateway. We can leave this as an area for further research.

REFERENCES

Bagal, P., Kalyanaraman, S., & Packer, B. (1999). Comparative study of RED, ECN and TCP rate control. *Technical Report.* USA: Rensselaer Polytechnic Institute.

Christiansen, M., Jeffay, K., Ott, D., & Smith, F.D. (2000). Tuning RED for web Traffic. *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication Sweden,* 139-150.

Cisco IOS Quality of Service Solutions Configuration Guide, Release 12.2

Clark, D. D., & Fang, W. (1998). Explicit Allocation of Best Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking-TON, 6* (4), 362-373.

Çatalkaya, G. (2003). *Simulation of a Local Congestion Reducing Routing Strategy for Multidestination Network.* İzmir; Dokuz Eylül University, Engineering Faculty, Msc. Thesis.

Dah-Ming, C., & Jain R. (1989). Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems, 17* (1), 1-14.

Fall, K., & Floyd, S. (1996). Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *ACM SIGCOMM Computer Communication Review*, *26* (3), 5-21.

Feng, W., Kandlur, D., Saha, D., & Shin, K. (1999). A Self-Configuring RED Gateway. *Proceedings IEEE INFOCOM '99, 3,* 1320-1328.

Floyd, S., & Jacobson, V. (1993). Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking - TON, 1* (4), 397-413.

Floyd, S. (1994). TCP and Explicit Congestion Notification. *ACM SIGCOMM Computer Communication Review, 24* (5), 8-23.

Floyd, S., Arcia, A., Ros, D., & Iyengar, J. (February 2010). Adding Acknowledgement Congestion Control to TCP. *RFC 5690,* retrived March 2010 from http://tools.ietf.org/html/rfc5690.

Floyd, S., Henderson, T., & Gurtov, A. (April, 2004). *The New Reno Modification to TCP's Fast Recovery Algorithm. RFC 3782, Proposed Standard.* Retrived June 2008, from http://www.ietf.org/rfc/rfc3782.txt.

Gerla, M., & Kleinrock, L. (1980). Flow control: a comparative survey. *IEEE Transactions on Communications*, *28* (4), 553-574.

Golubchik L., & Lui, J.C.S. (2002). Bounding of Performance Measures for a Threshold-based Queueuing System with Hysterisis. *IEEE Transactions on Computers,* 51 (4), 353-372.

Gupta N., & Mishra, G.D. (2009). Performance Analysis of an M/M/1/K Queue with Non-preemptive Priority. *International J. of Math. Sci. & Engg. Appls. (IJMSEA) ISSN 0973-9424*, *3* (2), *191-197.*

Hollot, C. V., Misra, V., Towsley, D.F., & Gong, W. (2001). On Designing Improved Controllers for AQM Routers Supporting TCP Flows. *Proceedings IEEE INFOCOM 2001, 3,* 1735-1744.

Internet Systems Consortium. (n.d.). Retrieved December 20, 2009, from http://www.isc.org

Kinicki, R., & Zheng, Z. (2001). A Performance Study of Explicit Congestion Notification (ECN) with Heterogeneous TCP Flows. *Networking-ICN 2001, 1,* 98-106.

Kleinrock, L. (Ed.). (1975). *Queueuing Systems.* Vol I. *Wiley.*

Kuzmanovic, A., Mondal, A., Floyd, S., & Ramakrishnan, K. K. (June, 2009). *Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK. RFC 5562, Experimental.* Retrived August 2009, from http://tools.ietf.org/html/rfc5562.

Lin, D., & Morris, R. (1997). Dynamics of random early detection. *Proceedings of the ACM SIGCOMM '97,* 127-137.

Lin, W., & Kumar P.R. (1984). Optimal Control of a Queueing System with Two Heterogeneous Servers, *IEEE Transactions on Automatic Control, 29* (8), 471-488.

Mathis, M., & Mahdavi, J. (1996). Forward Acknowledgment (FACK): Refining TCP Congestion Control. *Proceedings of SIGCOMM'96*, 281-191.

Mathis, M., & Mahdavi, J., & Floyd S., & Romanow, A. (1996). TCP Selective Acknowledgment Options. *RFC 2018, Proposed Standard, April 1996.*

Morrison, J. (1990). Two Server Queue with One Server Idle below a Threshold. *Queueing Systems: Theory and Applications, 7* (3-4), 325-336.

Nagle, J. (1984). Congestion control in IP/TCP internetworks. *ACM SIGCOMM Computer Communication Review, 14* (4), 11-17.

Ott, T. J., Lakshman, T. V., & Wong, L. H. (1999). SRED: Stabilized RED. *Proceedings IEEE INFOCOM'99, 3*, 1346-1355.

Padhye, J., Firoiu, V., Towsley, D., & Kurose J. (1998). Modeling TCP Thrughput: A Simple Model and its Empirical Validation. *Computer Communication Review, a publication of ACM SIGCOMM, 28* (4), 303-314.

Pan, R., Prabhakar, B., & Psounis, K. (2000). CHOK, A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation. *Proceedings IEEE INFOCOM 2000, 2,* 942-951.

Pentikousis, K. (2004). *Application/Traffic/Poisson - Poisson traffic generator for ns-2, ns-2 module.* Retrived June 2010, from http://ipv6.willab.fi/kostas/src/Application-Traffic-Poisson/

Postel, J. (1981). *Transmission control protocol-darpa internet program protocol specification, RFC793*. February 2009, http://www.ietf.org/rfc/rfc793.txt.

Ramakrishnan, K. K., & Jain R. (1988). A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer.

Applications, Technologies, Architectures, and Protocols for Computer Communication Symposium, 303-313.

Ramakrishnan, K.K., & Jain, R. (1990). A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Transactions on Computer Systems*, *8* (2), 158-181.

Ramakrishnan, K., Floyd, S., & Black, D. (September, 2001). *The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Proposed Standard*. Retrived May 2008 from http://www.rfc-editor.org/rfc/rfc3168.txt.

Rusmin, P.H., Machbub, C., & Harsoyo, A. (2007). White: Controlling The Internet Congestion With Integral Controller. *Control, Automation, Robotics and Vision,* 1, 1-6.

Savage, S., Anderson, T., Aggarwal, A., Becker, D., Cardwell, N., Collins, A., Hoffman, E., Snell, J., Vahdat, A., Voelker, G., & Zahorjan, J. (1999). Detour: informed Internet routing and transport. *IEEE Micro, 19* (1), 50–59.

Sterbenz, J. P. G., Krishnan, R., Hain, R.R., Jackson, A.W., Levin, D., Ramanathan, R., & Zao, J. (2002). Survivable mobile wireless networks: issues, challenges, and research directions. *Proceedings of the 1st ACM workshop on Wireless security,* 31-40.

Şiş, M. K. (1994). *A Dynamic Local Congestion Reducing Strategy Based on a Mini-Max Criterion*. USA; Polytechnic Institute of New York University, Ph.D. Thesis.

APPENDICES

A. Full Source Code of Orange Algorithm: orange.cc & orange.h

Source code of Orange Algorithm.

```
#ifndef orange_h
#define orange_h

#include <string.h>
#include "queue.h"
#include "config.h"

class Orange : public Queue {
	public:
		Orange();
		~Orange() {
			delete q_;
		}
	protected:
		void reset();
		int command(int argc, const char*const* argv);
		void enque(Packet*);
		Packet* deque();
		PacketQueue *q_; /*Underlying FIFO queue*/
		int drop_front_;
		int summarystats;
		void print_summarystats();
		int qib_;
		int mean_pktsize_;
		int orange_limit;
		int queue_limit;
		double orange_timer;
		double ExpOrangeTimer;
		double lastDrop;
		int byOrange;
		int byDroptail;
};

#endif
/*------------------------------------------------------------------*/

#ifndef lint
static const char rcsid[] =
  "@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/queue/orange.cc,v 1.15 2003/01/16 19:02:54 sfloyd Exp $
(LBL)";
#endif


#include "orange.h"
#include "flags.h"
#include "basetrace.h"
#include "hdr_qs.h"
#include "random.h"

static  class OrangeClass : public TclClass {
	public:
		OrangeClass() : TclClass("Queue/Orange") {}
		TclObject* create(int, const char*const*) {
```

```
                        return (new Orange);
                }
} class_orange;

Orange::Orange(){
        q_ = new PacketQueue;
        pq_ = q_;

        bind_bool("drop_front_", &drop_front_);
        bind_bool("summarystats_", &summarystats);
        bind_bool("queue_in_bytes_", &qib_);
        bind("mean_pktsize_", &mean_pktsize_);
        bind("orange_limit_", &orange_limit);
        bind("queue_limit_", &queue_limit);
        bind("orange_timer_", &orange_timer);

        //printf("Timer: %f\n", orange_timer);

}

void Orange::reset(){
        Queue::reset();
}

int Orange::command(int argc, const char*const* argv) {
        if (argc == 2) {
                if (strcmp(argv[1], "printstats") == 0) {
                        print_summarystats();
                        return (TCL_OK);
                }
        }

        if (argc == 3) {
                if (!strcmp(argv[1], "packetqueue-attach")) {
                        delete q_;
                        if (!(q_ = (PacketQueue*) TclObject::lookup(argv[2]))) {
                                return (TCL_ERROR);
                        }
                        else {
                                pq_ = q_;
                                return (TCL_OK);
                        }
                }
        }

        return Queue::command(argc, argv);
}

void Orange::enque(Packet* p) {

        if (summarystats) {
                Queue::updateStats(qib_?q_->byteLength():q_->length());
        }

        int qlimBytes  = queue_limit  * mean_pktsize_;
        int qlimBytes2 = orange_limit * mean_pktsize_;

        if ((!qib_ && (q_->length() + 1) >= queue_limit) ||
                ( qib_ && (q_->length() + hdr_cmn::access(p)->size()) >= qlimBytes)){

                //if the queue would overflow if we added this packet...
                if (drop_front_) {  /*remove from head of queue*/
                        q_->enque(p);
                        Packet *pp = q_->deque();
```

```
					drop(pp);
					byDroptail++;
				}
				else {
					drop(p);
					//printf("DROP!");
					byDroptail++;
				}
			}
			else {
				if ((!qib_ && (q_->length() + 1) >= orange_limit) ||
			( qib_ && (q_->length() + hdr_cmn::access(p)->size()) >= qlimBytes2)){

					double now = Scheduler::instance().clock();
					double differ = now - lastDrop;
					//printf("\nnow-%5.3f ",now");
					//printf("\tlastdroptime-%5.3f ", lastDrop);
					//printf("\tdifference-%5.3f ", differ);
					//printf("\torangetimer-%5.3f ", orange_timer/1000);

					//double ExpOrangeTimer = orange_timer;
					//printf("ExpOrangeTimer: %f\n", ExpOrangeTimer);

					if ((differ >= ExpOrangeTimer/1000) || lastDrop == 0) {
						lastDrop = now;
						ExpOrangeTimer = Random::exponential (orange_timer);
						if (drop_front_) {
							q_->enque(p);
							Packet *pp = q_->deque();
							drop(pp);
							byOrange++;
						}
						else {
							drop(p);
							byOrange++;
						}
						//print("DROP!");
					}
					else {
						q_->enque(p);
					}
				}
				else {
					q_->enque(p);
				}
			}

			if (byOrange && byDroptail) {
				//printf("\nNumber of dropped packets by droptail: %d", byDroptail);
				//printf("\nby %d threshold and %5.3fms timer: %d", orange_limit, orange_timer, byOrange);
				//printf("\n-------------------------------------------------\n");
			}
}

Packet* Orange::deque(){

			if (summarystats && &Scheduler::instance() != NULL) {
				Queue::updateStats(qib_?q_->byteLength():q_->length());
			}
			return q_->deque();
}

void Orange::print_summarystats(){
			double now = Scheduler::instance().clock();
```

```
        printf("True average queue: %5.3f", true_ave_);
        printf("\nNumber of dropped packets by %d threshold: %d", orange_limit, byOrange);
        printf("\nNumber of dropped packets by droptail: %d", byDroptail);
        if (qib_) {
                printf(" (in bytes)");
        }
        printf("\ntime: %5.3f\n", total_time_);
}
```

B. Source Code of the Function: Drop_Early_Orange

The aim of the function is to decide if the incoming packet to the queue will be dropped or not according to the busy – idle state of the orange timer.

```
int ORANGEQueue::drop_early_orange(Packet* pkt)
{

double now = Scheduler::instance().clock();
double differ = now - lastDrop;


if ((differ >= (orange_timer_/1000)) || lastDrop == 0) {
{
        lastDrop = now;
        expOrangeTimer = Random::exponential (orange_timer_);

        byOrange++;

        return (1); //DROP
        }


return (0);                             // no DROP/mark Alternate server is busy
}
```

C. Source Code of the Function: enqueue

The aim of the function is to enqueue the incoming packet according to the algorithm applied.

```
void ORANGEQueue::enque(Packet* pkt)
{
        hdr_cmn* ch = hdr_cmn::access(pkt);
        ++edv_.count;
        edv_.count_bytes += ch->size();

        register double qavg = edv_.v_ave;
        int droptype = DTYPE_NONE;
        int qlen = qib_ ? q_->byteLength() : q_->length();
        int qlim = qib_ ? (qlim_ * edp_.mean_pktsize) : qlim_;
```

```
		if (qlen >= edp_.th_min && drop_early_orange(pkt)) {

				droptype = DTYPE_UNFORCED;

		} else {
			/* No packets are being dropped.  */
			edv_.v_prob = 0.0;
			edv_.old = 0;
		}
		if (qlen >= qlim) {
			// see if we've exceeded the queue size
			droptype = DTYPE_FORCED;
		}

		if (droptype == DTYPE_UNFORCED) {

			if (pkt_to_drop != pkt) {
				q_->enque(pkt);
				q_->remove(pkt_to_drop);
				pkt = pkt_to_drop;
}
			if (de_drop_ != NULL) {

			if (EDTrace != NULL)
					((Trace *)EDTrace)->recvOnly(pkt);

				reportDrop(pkt);
				de_drop_->recv(pkt);
			}
			else {
				reportDrop(pkt);
				drop(pkt);
			}
		} else {
			/* forced drop, or not a drop: first enqueue pkt */
			q_->enque(pkt);

			/* drop a packet if we were told to */
			if (droptype == DTYPE_FORCED) {
				/* drop random victim or last one */
				pkt = pickPacketToDrop();
				q_->remove(pkt);
				reportDrop(pkt);
				drop(pkt);
				if (!ns1_compat_) {
					// bug-fix from Philip Liu, <phill@ece.ubc.ca>
					edv_.count = 0;
					edv_.count_bytes = 0;
				}
			}
		}

		double now = Scheduler::instance().clock();
 if (droptype == DTYPE_FORCED)
 printf("%10f ByTailDrop\n", now);


 if (droptype == DTYPE_UNFORCED)
 printf("%10f ByOrange\n", now);

		return;
}
```

D. Queue Size Script: queueSize.awk

This awk script produces a trace file of queue size over time from the output trace of NS.

```
BEGIN {
 queueSize = 0
 prevTime = 0
 prevQSize = 0
}

{
	# Trace line format: normal
	if ($2 != "-t") {
		event = $1
		time = $2
		if (event == "+" || event == "-") node_id = $3
		if (event == "r" || event == "d") node_id = $4
		flow_id = $8
		pkt_id = $12
		pkt_size = $6
		flow_t = $5
		level = "AGT"
	}
	# Trace line format: new
	if ($2 == "-t") {
		event = $1
		time = $3
		node_id = $5
		flow_id = $39
		pkt_id = $41
		pkt_size = $37
		flow_t = $45
		level = $19
	}

	WillPrint = 0

	# Update total received packets' size and store packets arrival time
	if (level == "AGT" && node_id == src && event == "+"){
 queueSize++
 WillPrint = 1
 }

	if (level == "AGT" && node_id == src && event == "-"){
 queueSize--
 WillPrint = 1
 }

 if (level == "AGT" && node_id == dst && event == "d"){
	queueSize--
 WillPrint = 1
 }

 if (WillPrint == 1)
 printf("%10g %10g\n", time, queueSize)

}

END {
```

}

E. Simulation Statistics Script: avgStats.awk

This awk script produces statistical information about simulation from the output trace of NS.

```
BEGIN {
        recvdSize = 0
        startTime = 1e6
        stopTime = 0
}

{
        # Trace line format: normal
        if ($2 != "-t") {
                event = $1
                time = $2
                if (event == "+" || event == "-") node_id = $3
                if (event == "r" || event == "d") node_id = $4
                flow_id = $8
                pkt_id = $12
                pkt_size = $6
                flow_t = $5
                level = "AGT"
        }
        # Trace line format: new
        if ($2 == "-t") {
                event = $1
                time = $3
                node_id = $5
                flow_id = $39
                pkt_id = $41
                pkt_size = $37
                flow_t = $45
                level = $19
        }

        # Store packets send time
        if (level == "AGT" && flow_id == flow && node_id == src &&
           sendTime[pkt_id] == 0 && (event == "+" || event == "s") && pkt_size >= pkt) {
                if (time < startTime) {
                        startTime = time
                }
                sendTime[pkt_id] = time
                this_flow = flow_t
        }

        # Update total received packets' size and store packets arrival time
        if (level == "AGT" && flow_id == flow && node_id == dst &&
           event == "r" && pkt_size >= pkt) {
                if (time > stopTime) {
                        stopTime = time
                }
                # Rip off the header
                hdr_size = pkt_size % pkt
                pkt_size -= hdr_size
                # Store received packet's size
                recvdSize += pkt_size
                # Store packet's reception time
                recvTime[pkt_id] = time
        }
```

```
}

END {
	# Compute average delay
	delay = avg_delay = recvdNum = 0
	for (i in recvTime) {
		if (sendTime[i] == 0) {
			printf("\nError in delay.awk: receiving a packet that wasn't sent %g\n",i)
		}
		delay += recvTime[i] - sendTime[i]
		recvdNum ++
	}
	if (recvdNum != 0) {
		avg_delay = delay / recvdNum
	} else {
		avg_delay = 0
	}

	# Compute average jitters
	jitter1 = jitter2 = jitter3 = jitter4 = jitter5 = 0
	prev_time = delay = prev_delay = processed = deviation = 0
	prev_delay = -1
	for (i=0; processed<recvdNum; i++) {
		if(recvTime[i] != 0) {
			if(prev_time != 0) {
				delay = recvTime[i] - prev_time
				e2eDelay = recvTime[i] - sendTime[i]
				if(delay < 0) delay = 0
				if(prev_delay != -1) {
					jitter1 += abs(e2eDelay - prev_e2eDelay)
					jitter2 += abs(delay-prev_delay)
					jitter3 += (abs(e2eDelay-prev_e2eDelay) - jitter3) / 16
					jitter4 += (abs(delay-prev_delay) - jitter4) / 16
				}
#				deviation += (e2eDelay-avg_delay)*(e2eDelay-avg_delay)
				prev_delay = delay
				prev_e2eDelay = e2eDelay
			}
			prev_time = recvTime[i]
			processed++
		}
	}
	if (recvdNum != 0) {
		jitter1 = jitter1*1000/recvdNum
		jitter2 = jitter2*1000/recvdNum
	}
#	if (recvdNum > 1) {
#		jitter5 = sqrt(deviation/(recvdNum-1))
#	}

	# Output
	if (recvdNum == 0) {

	printf("####################################################################\n" \
			"#  Warning: no packets were received, simulation may be too short  #\n" \

"####################################################################\n\n")
	}
	printf("\n")
	printf(" %15s:  %g\n", "flowID", flow)
	printf(" %15s:  %s\n", "flowType", this_flow)
	printf(" %15s:  %d\n", "srcNode", src)
	printf(" %15s:  %d\n", "destNode", dst)
```

```
        printf(" %15s:  %d\n", "startTime", startTime)
        printf(" %15s:  %d\n", "stopTime", stopTime)
        printf(" %15s:  %g\n", "receivedPkts", recvdNum)
        printf(" %15s:  %g\n", "avgTput[kbps]", (recvdSize/(stopTime-startTime))*(8/1000))
        printf(" %15s:  %g\n", "avgDelay[ms]", avg_delay*1000)
        printf(" %15s:  %g\n", "avgJitter1[ms]", jitter1)
        printf(" %15s:  %g\n", "avgJitter2[ms]", jitter2)
        printf(" %15s:  %g\n", "avgJitter3[ms]", jitter3*1000)
        printf(" %15s:  %g\n", "avgJitter4[ms]", jitter4*1000)
#       printf(" %15s:  %g\n", "avgJitter5[ms]", jitter5*1000)

#       %9s %4s %4s %6s %5s %13s %14s %13s %15s %15s %15s %15s %15s\n\n",          \
#           "flow","flowType","src","dst","start","stop","receivedPkts",           \
#           "avgTput[kbps]","avgDelay[ms]","avgJitter1[ms]","avgJitter2[ms]",      \
#           "avgJitter3[ms]","avgJitter4[ms]","avgJitter5[ms]")
#       printf("  %6g %9s %4d %4d %6d %5d %13g %14s %13s %15s %15s %15s %15s\n\n", \
#           flow,this_flow,src,dst,startTime, stopTime, recvdNum,                  \
#           (recvdSize/(stopTime-startTime))*(8/1000),avg_delay*1000,              \
#           jitter1,jitter2,jitter3*1000,jitter4*1000,jitter5*1000)
}

function abs(value) {
        if (value < 0) value = 0-value
        return value
}
```

F. Instant Throughput Script: instantThroughput.awk

This awk script produces a trace file of throughput over time from the output trace of NS.

```
BEGIN {
        recv = 0
        currTime = prevTime = 0
        printf("# %10s %10s %5s %5s %15s %18s\n\n", \
            "flow","flowType","src","dst","time","throughput")
}

{
        # Trace line format: normal
        if ($2 != "-t") {
                event = $1
                time = $2
                if (event == "+" || event == "-") node_id = $3
                if (event == "r" || event == "d") node_id = $4
                flow_id = $8
                pkt_id = $12
                pkt_size = $6
                flow_t = $5
                level = "AGT"
        }
        # Trace line format: new
        if ($2 == "-t") {
                event = $1
                time = $3
                node_id = $5
                flow_id = $39
```

```
                pkt_id = $41
                pkt_size = $37
                flow_t = $45
                level = $19
        }

        # Init prevTime to the first packet recv time
        if(prevTime == 0)
                prevTime = time

        # Calculate total received packets' size
        if (level == "AGT" && flow_id == flow && node_id == dst &&
          event == "r" && pkt_size >= pkt) {
                # Rip off the header
                hdr_size = pkt_size % pkt
                pkt_size -= hdr_size
                # Store received packet's size
                recv += pkt_size
                # This 'if' is introduce to obtain clearer
                # plots from the output of this script
                if((time - prevTime) >= tic*10) {
                        printf("  %10g %10s %5d %5d %15g %18g\n", \
                                flow,flow_t,src,dst,(prevTime+1.0),0)
                        printf("  %10g %10s %5d %5d %15g %18g\n", \
                                flow,flow_t,src,dst,(time-1.0),0)
                }
                currTime += (time - prevTime)
                if (currTime >= tic) {
                        printf("  %10g %10s %5d %5d %15g %18g\n", \
                            flow,flow_t,src,dst,time,(recv/currTime)*(8/1000))
                        recv = 0
                        currTime = 0
                }
                prevTime = time
        }
}

END {
        printf("\n\n")
}
```

G. Script for Topology Alpha

The aim of this script is to construct the M/M/1/K queue Topology Alpha. In order to connect 1000 poisson sources to a node, we use 4 different nodes as at most 250 sources can be connected to a node.

```
#Create a simulator object
set ns [new Simulator]

#Open the nam trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Open the trace file
```

```
set nftr [open out.tr w]
$ns trace-all $nftr

#Define a 'finish' procedure
proc finish {} {
        global ns nf nftr file_0 file_1
        close $nf
        close $nftr
        close $file_0
        close $file_1

  set parse {
                {
                  if ($6 == "cwnd_") {
                                print $1, $7;
                  }
                }}
        exec awk $parse cwnd0.tr > xcwnd0.tr
        exec awk $parse cwnd1.tr > xcwnd1.tr

  #exec /ns/xgraph xcwnd0.tr &
  #exec nam out.nam &
  exit 0
}

set pkt 0
set run 1000

# seed the default RNG
global defaultRNG
$defaultRNG seed 9999

# create the RNGs and set them to the correct substream
set arrivalRNG [new RNG]
set sizeRNG [new RNG]

for {set k 1 } {$k <= $run} {incr k} {
  $arrivalRNG next-substream
  $sizeRNG next-substream
}


set pktsize [new RandomVariable/Exponential]
$pktsize set avg_ 100
$pktsize use-rng $sizeRNG

set S [$ns node]
set D [$ns node]

$ns duplex-link $S $D 1Mb 10ms RED
$ns queue-limit $S $D 10000

for {set i 1} {$i <= 4} {incr i} {
  set s($i) [$ns node]
  set d($i) [$ns node]

  $ns duplex-link $s($i) $S 100Mb 0ms DropTail
  $ns duplex-link $D $d($i) 100Mb 0ms DropTail

  for {set j 1} {$j <= 250} {incr j} {
    set udp($i,$j) [new Agent/UDP]
    $ns attach-agent $s($i) $udp($i,$j)

    set null($i,$j) [new Agent/Null]
```

```
    $ns attach-agent $d($i) $null($i,$j)

    $ns connect $udp($i,$j) $null($i,$j)

    set poisson($i,$j) [new Application/Traffic/Poisson]
    $poisson($i,$j) attach-agent $udp($i,$j)

    $poisson($i,$j) set interval_ [expr 1.0/1.2]


    set pkt  [expr round([$pktsize value])]
    #puts  "$i, $j:    $pkt"
    $poisson($i,$j) set packetSize_ $pkt
    #$poisson($i,$j) set packetSize_ 1000

    $ns at 0 "$poisson($i,$j) start"
  }
}

#Call the finish procedure
$ns at 10.0 "finish"

#Run the simulation
$ns run
```

H. Script for Topology Bravo

The aim of this script is to construct Topology Bravo.

```
#Create a simulator object
set ns [new Simulator]

#Open the trace file
set nftr [open out.tr w]
$ns trace-all $nftr

#Define a 'finish' procedure
proc finish {} {
        global ns nf nftr file_0 file_1
        close $nf
        close $nftr
        close $file_0
        close $file_1
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]

#Create links between the nodes
$ns duplex-link $n2 $n0 10Mb 0ms DropTail
$ns duplex-link $n0 $n1 1Mb 0ms ORANGE

#$ns queue-limit $n3 $n4 5
$ns queue-limit $n0 $n1 120

#Setup a TCP connection nd attach it to node n0
set tcp0 [new Agent/TCP/Reno]
$tcp0 set packetSize_ 1000
$tcp0 set class_ 1
```

```
$ns attach-agent $n2 $tcp0
$tcp0 set window_ 500

#Setup a FTP over TCP connection
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0
$ftp0 set type_ FTP

#Create a Null agent (a traffic sink) and attach it to node n4
set null0 [new Agent/TCPSink]
$ns attach-agent $n1 $null0
$ns connect $tcp0 $null0


$ns at 0 "$ftp0 produce 10000"

#Call the finish procedure
$ns at 100.0 "finish"

#Run the simulation
$ns run
```

I. Script for Topology Charlie

The aim of this script is to construct Topology Charlie.

```
#Create a simulator object
set ns [new Simulator]

#Define different colors for data flows
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green

#Open the nam trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Open the trace file
set nftr [open out.tr w]
$ns trace-all $nftr

# Tracing cwnd of TCP Agents
set file_0 [open cwnd0.tr w]
set file_1 [open cwnd1.tr w]

#Define a 'finish' procedure
proc finish {} {
	global ns nf nftr file_0 file_1
	close $nf
	close $nftr
	close $file_0
	close $file_1

  set parse {
		{
		   if ($6 == "cwnd_") {
				print $1, $7;
		   }
		}}
	exec awk $parse cwnd0.tr > xcwnd0.tr
```

```
    exec awk $parse cwnd1.tr > xcwnd1.tr

  exec /ns/xgraph xcwnd0.tr &
  #exec nam out.nam &
  exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n2 10Mb 10ms DropTail
$ns duplex-link $n1 $n2 10Mb 10ms DropTail
$ns duplex-link $n2 $n3 1Mb 10ms RED

$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right

#Monitor the queue for the link between node 2 and node 3
$ns duplex-link-op $n2 $n3 queuePos 0.5

#$ns queue-limit $n3 $n4 5
$ns queue-limit $n2 $n3 120


#Node0
#Setup a TCP connection nd attach it to node n0
set tcp0 [new Agent/TCP/Reno]
$tcp0 set packetSize_ 1000
$tcp0 set class_ 1
$ns attach-agent $n0 $tcp0
#$tcp0 set window_ 500

#Setup a FTP over TCP connection
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0
$ftp0 set type_ FTP

#Node1
#Setup a TCP connection nd attach it to node n1
set tcp1 [new Agent/TCP/Reno]
$tcp1 set packetSize_ 1000
$tcp1 set class_ 2
$ns attach-agent $n1 $tcp1
#$tcp1 set window_ 500

#Setup a FTP over TCP connection
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$ftp1 set type_ FTP

#Create a Null agent (a traffic sink) and attach it to node n4
set null0 [new Agent/TCPSink]
$ns attach-agent $n3 $null0

set null1 [new Agent/TCPSink]
$ns attach-agent $n3 $null1

#Connect the traffic sources with the traffic sink
$ns connect $tcp0 $null0
```

```
$ns connect $tcp1 $null1

$tcp0 set fid_ 1
$tcp1 set fid_ 2

#Trace cwnd_
$tcp0 trace cwnd_
$tcp1 trace cwnd_
$tcp0 trace rtt_
$tcp1 trace rtt_
$tcp0 trace ssthresh_
$tcp1 trace ssthresh_

$tcp0 attach $file_0
$tcp1 attach $file_1


#Schedule events for the CBR agents
#$ns at 0 "$ftp0 start"
#$ns at 20 "$ftp1 start"

#$ns at 200 "$ftp0 stop"
#$ns at 200 "$ftp1 stop"

$ns at 0 "$ftp0 produce 10000"
$ns at 20 "$ftp1 produce 10000"

#$ftp0 produce 100


#Call the finish procedure
$ns at 200.0 "finish"

#Run the simulation
$ns run
```

J. Script for Topology Delta

The aim of this script is to construct Topology Delta.

```
#Create a simulator object
set ns [new Simulator]

#Define different colors for data flows
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green

#Open the nam trace file
set nf [open out.tr w]
$ns trace-all $nf

#Define a 'finish' procedure
proc finish {} {
        global ns nf file_0 file_1
        close $nf
        close $file_0
        close $file_1

  set parse {
            {
               if ($6 == "cwnd_") {
```

```
                    print $1, $7;
            }
         }}
     exec awk $parse cwnd0.tr > xcwnd0.tr
     exec awk $parse cwnd1.tr > xcwnd1.tr

  exec xgraph -bb -tk -m xcwnd0.tr xcwnd1.tr &
  exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n3 10Mb 10ms DropTail
$ns duplex-link $n1 $n3 10Mb 10ms DropTail
$ns duplex-link $n2 $n3 10Mb 10ms DropTail
$ns duplex-link $n3 $n4  1Mb 20ms DropTail


#$ns queue-limit $n3 $n4 5

#set monitor [$ns monitor-queue $n3 $n4 stdout 0.1]
#[$ns link $n3 $n4] queue-sample-timeout;
#[$ns link $S $D] start-tracing

#$ns trace-queue $n3 $n4 [open Q3.tr w]
$ns duplex-link-op $n0 $n3 orient right-down
$ns duplex-link-op $n1 $n3 orient right
$ns duplex-link-op $n2 $n3 orient right-up
$ns duplex-link-op $n3 $n4 orient right

#Monitor the queue for the link between node 2 and node 3
$ns duplex-link-op $n3 $n4 queuePos 0.5

# Tracing a queue
#$ns queue-limit $n3 $n4 5000
#set Queue/Orange orange_limit 200
#set Queue/Orange queue_limit  300

#Node 0
#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$udp0 set class_ 1
$ns attach-agent $n0 $udp0
# Create a CBR traffic source and attach it to udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0



#Node1
#Setup a TCP connection nd attach it to node n1
set tcp0 [new Agent/TCP/Reno]
$tcp0 set packetSize_ 1000
$tcp0 set class_ 2
$ns attach-agent $n1 $tcp0
#Setup a FTP over TCP connection
```

```
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0
$ftp0 set type_ FTP

#Node2
#Setup a TCP connection nd attach it to node n2
set tcp1 [new Agent/TCP/Reno]
$tcp1 set packetSize_ 1000
$tcp1 set class_ 3
$ns attach-agent $n2 $tcp1
#Setup a FTP over TCP connection
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$ftp1 set type_ FTP

#Create a Null agent (a traffic sink) and attach it to node n4
set null0 [new Agent/Null]
$ns attach-agent $n4 $null0

set sink0 [new Agent/TCPSink]
$ns attach-agent $n4 $sink0

set sink1 [new Agent/TCPSink]
$ns attach-agent $n4 $sink1


#Connect the traffic sources with the traffic sink
$ns connect $udp0 $null0
$ns connect $tcp0 $sink0
$ns connect $tcp1 $sink1
$tcp0 set fid_ 2
$tcp1 set fid_ 3

# Tracing cwnd of TCP Agents
set file_0 [open cwnd0.tr w]
set file_1 [open cwnd1.tr w]
$tcp0 trace cwnd_
$tcp1 trace cwnd_
$tcp0 trace rtt_
$tcp1 trace rtt_
$tcp0 trace ssthresh_
$tcp1 trace ssthresh_
$tcp0 attach $file_0
$tcp1 attach $file_1


#Schedule events for the CBR agents
$ns at 0.10 "$cbr0 start"
$ns at 0.20 "$ftp0 start"
$ns at 0.20 "$ftp1 start"

$ns at 4.70 "$ftp1 stop"
$ns at 4.70 "$ftp0 stop"
$ns at 4.70 "$cbr0 stop"


#Call the finish procedure
$ns at 5.0 "finish"

#Run the simulation
$ns run
```

K. Script for Topology Echo

The aim of this script is to construct Topology Echo. Number of flow groups, number of flows in a group, and aggregate bandwidth are parameters to this script.

```
#Create a simulator object
set ns [new Simulator]

set nn    3
set fn   20
set AggBW 90

#Open the nam trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Open the trace file
set nftr [open out.tr w]
$ns trace-all $nftr

#Define a 'finish' procedure
proc finish {} {
	global ns nf nftr file_0


	close $file_0
	close $nf
	close $nftr

	set parse {
		{
		  if ($6 == "cwnd_") {
			print $1, $7;
		  }
		}}

	exec awk $parse cwnd0.tr > xcwnd0.tr

  #exec xgraph -bb -tk -m xcwnd0.tr xcwnd1.tr &
  #exec nam out.nam &
  exit 0
}

#Create the nodes
set n3 [$ns node]
set n4 [$ns node]
$ns duplex-link $n3 $n4 10Mb 5ms ORANGE

$ns duplex-link-op $n3 $n4 orient right

#set redqueue [$link $n3 $n4]
#Queue/ORANGE set thresh_ 3
#Queue/ORANGE set maxthresh_ 30

$ns queue-limit $n3 $n4 120

#Monitor the queue for the link between node 3 and node 4
$ns duplex-link-op $n3 $n4 queuePos 0.5
#set monitor [$ns monitor-queue $n3 $n4 stdout 0.1]
```

```
for {set i 0} {$i < $nn} {incr i} {
   for {set j 0} {$j < $fn} {incr j} {

   set n($i,$j) [$ns node]

   $ns duplex-link $n($i,$j) $n3 [expr $AggBW/($nn*$fn)]Mb [expr (pow(2,2-$i)*50-10)/2]ms DropTail

   #Setup a TCP connection nd attach it to node n1
   set tcp($i,$j) [new Agent/TCP/Reno]
   $tcp($i,$j) set packetSize_ 1000
   $ns attach-agent $n($i,$j) $tcp($i,$j)
   #Setup a FTP over TCP connection
   set ftp($i,$j) [new Application/FTP]
   $ftp($i,$j) attach-agent $tcp($i,$j)
   $ftp($i,$j) set type_ FTP

   set sink($i,$j) [new Agent/TCPSink]
   $ns attach-agent $n4 $sink($i,$j)

   #Connect the traffic sources with the traffic sink
   $ns connect $tcp($i,$j) $sink($i,$j)

   if {$i == 0} {
     $ns color [expr $fn *  $i + $j] Blue
     $tcp($i,$j) set class_ 0
     $tcp($i,$j) set fid_ [expr $fn *  $i + $j]
   }
   if {$i == 1} {
     $ns color [expr $fn *  $i + $j] Red
     $tcp($i,$j) set class_ 1
     $tcp($i,$j) set fid_ [expr $fn *  $i + $j]
   }
   if {$i == 2} {
     $ns color [expr $fn *  $i + $j] Green
     $tcp($i,$j) set class_ 2
     $tcp($i,$j) set fid_ [expr $fn *  $i + $j]
   }

   #Schedule events for the CBR agents
   if {$j < [expr $fn/2]} {
     $ns at 0.00 "$ftp($i,$j) start"
   }
   if {$j >= [expr $fn/2]} {
     $ns at 2.00 "$ftp($i,$j) start"
   }

   $ns at 100.00 "$ftp($i,$j) stop"
  }
}



#Call the finish procedure
$ns at 100.0 "finish"

#Run the simulation
$ns run
```